MaxTouch Driver Architecture (HA)

South China
Pitter Liao
2022 Jan

ECN

v0.1a: based on driver version v4.11 (Jan 07/2022)

Hardware Specification

POR Sequence

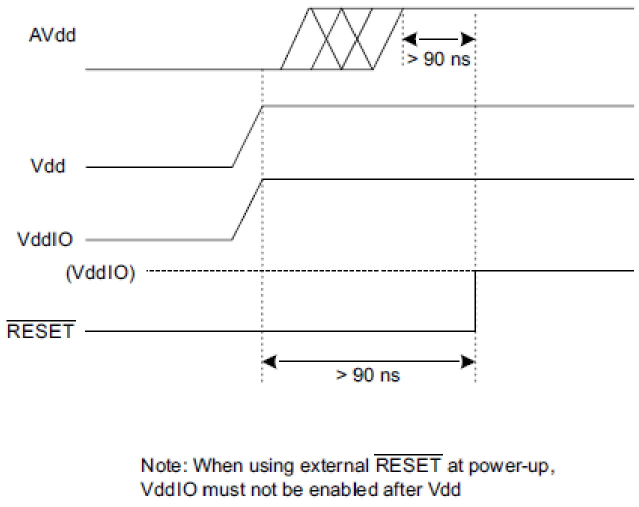
Power-on Reset

There is an internal Power-on Reset (POR) in the device.

If an external reset is to be used the device must be held in $\overline{\text{RESET}}$ (active low) while the digital (Vdd), analog (AVdd) and digital I/O (VddIO) power supplies are powering up. The supplies must have reached their nominal values before the $\overline{\text{RESET}}$ signal is deasserted (that is, goes high). This is shown in Figure 5-1. See Section 11.2 "Recommended Operating Conditions" for nominal values for the power supplies to the device.

A diode from AVDD to VDD is present in the device. If AVDD and VDD are driven from different supplies, the Vdd supply must be powered up earlier than AVdd.

FIGURE 5-1: POWER SEQUENCING ON THE MXT336UD-MAU001



It is recommended that customer designs include the capability for the host to control all the maXTouch power supplies and pull the $\overline{\text{RESET}}$ line low.

After power-up, the device typically takes 122 ms to 362 ms before it is ready to start communications, depending on the configuration.

In POR sequence, Reset line **must** keep low level(< 0.3 * VDDIO) until VDD reached valid voltage (~3.3v, see minimum VDD value in datasheet) and delay 90ns. Otherwise there is chance of POR failed.

Note, if your POR sequence is not matched, you should re-POR the device instead of asserting Reset\ Pin only.

Asserting the Reset\ Pin (without power rail) is only valid when POR successfully and later.

The HA chip Reset\ Pin is very important for retrieving the Seqnum, since after reset, the Sequm number will be 'Zero'.

Bootloader mode

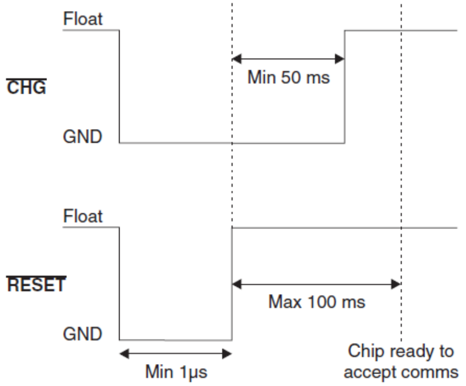
2.2 Command Processor Force-flash Sequence

Write 0xA5 to the Command Processor Object's RESET field to enter the bootloader mode. Refer to the *Protocol Guide* for your device for information on how to do this.

2.3 $\overline{\text{CHG}}$ and $\overline{\text{RESET}}$ Force-flash Sequence

With this sequence the $\overline{\text{CHG}}$ line is held low while the chip is powered up after a reset (see Figure 2-1).

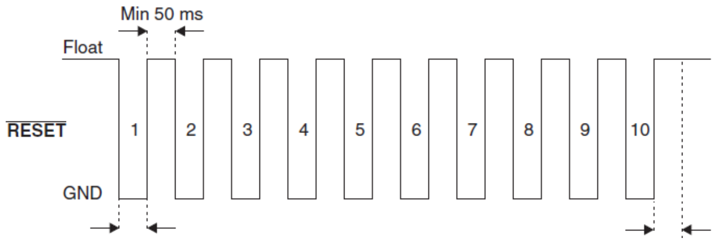
Figure 2-1. $\overline{\text{CHG}}$ and $\overline{\text{RESET}}$ Force-flash Sequence



2.4 $\overline{\text{RESET}}$ Toggling Force-flash Sequence

With this sequence the $\overline{\text{RESET}}$ line is asserted ten times in a row without communicating via the I^2C-compatible bus between the resets.

Figure 2-2. $\overline{\text{RESET}}$ Toggling Force-flash Sequence



Bootloader mode is **only** for flash the firmware content. Because the chip has firmware inside original and we don't need this action normally. But we should know what's bootloader mode and how to avoid to enter it.

We have 3 methods to get into bootloader mode:

1. Send T6 reset command with dedicated parameter(0xA5)
2. Stretched CHG line for more than 50ms when Reset line de-assert.
3. Toggling Reset line for more than 10 times without I2C communication.

So you know how to entered the boot loader mode now, while you know how to avoid enter it by mistake.

For exiting the bootloader mode, you could re-POR or sensor exit command through I2C.

You could refer the MXTAN0216_maXTouch_Bootloader document for more details.

I2C Specification

7.5 SDA and SCL

The I^2C bus transmits data and clock with SDA and SCL, respectively. These are open-drain. The device can only drive these lines low or leave them open. The termination resistors (Rp) pull the line up to VddIO if no I^2C device is pulling it down.

The termination resistors should be chosen so that the rise times on SDA and SCL meet the I^2C specifications for the interface speed being used, bearing in mind other loads on the bus. For best latency performance, it is recommended that no other devices share the I^2C bus with the maXTouch controller.

7.6 Clock Stretching

The device supports clock stretching in accordance with the I^2C specification. It may also instigate a clock stretch if a communications event happens during a period when the device is busy internally. The maximum clock stretch is 2 ms and typically less than 350 µs.

11.9 I^2C Specification

Parameter	Value
Address	0x4A
I^2C specification (1)	Revision 6.0
Maximum bus speed (SCL) (2)	1 MHz
Standard Mode (3)	100 kHz
Fast Mode (3)	400 kHz
Fast Mode Plus (3)	1 MHz

Note 1: More detailed information on I^2C operation is available from www.nxp.com/documents/user_manual/UM10204.pdf.
2: In systems with heavily laden I^2C lines, even with minimum pull-up resistor values, bus speed may be limited by capacitive loading to less than the theoretical maximum.
3: The values of pull-up resistors should be chosen to ensure SCL and SDA rise and fall times meet the I^2C specification. The value required will depend on the amount of capacitance loading on the lines.

1. Clock Stretching must be supported by host controller
2. I2C wave should match open document --- “I2C-bus specification and user manual” by NXP

MICROCHIP

I2C Communication(HA)

<1> Writing the Normal Object

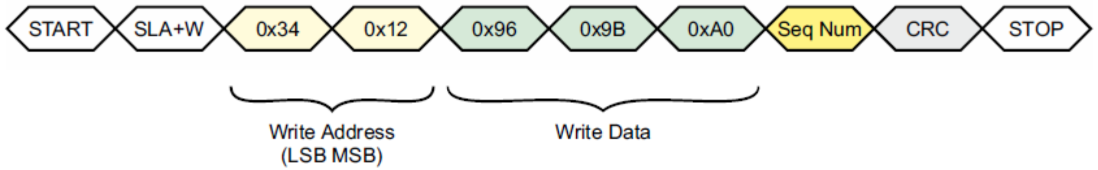
7.2 Writing To the Device

An I^{2}C WRITE cycle consists of the following bytes:

START	1 bit	I^{2}C START condition
SLA+W	1 byte	I^{2}C address of the device (see Section 7.1 "I2C Address")
Address (LSByte, MSByte)	2 bytes	Address of the location at which the data writing starts. This address is stored as the address pointer.
Data	0 .. 11 bytes	The actual data to be written. The data is written to the device, starting at the location of the address pointer. The address pointer returns to its starting value when the I^{2}C STOP condition is detected. Note that a maximum of 11 bytes of data can be written in any one transaction.
Sequence number	1 byte	The sequence number for this write. The sequence number must start at 0 for the first write after power-up/reset and incremented by 1 for each subsequent write. When the sequence number reaches 255, it is reset to 0.
CRC	1 byte	An 8-bit CRC that includes all the bytes that have been sent, including the two address bytes, but not the SLA+W byte. If the device detects an error in the CRC during a write transfer, a COMSERR fault is reported by the Command Processor T6 object.
STOP	1 bit	I^{2}C STOP condition

Figure 7-1 shows an example of writing three bytes of data to contiguous addresses starting at 0x1234.

FIGURE 7-1: EXAMPLE OF A THREE-BYTE WRITE STARTING AT ADDRESS 0x1234



The object writing operation finishes in one transmit cycle, the `Seqnum(W)` and `CRC` is required in data content.

The Touch controller asserting the ACK mostly means package is received and self `Seqnum(W)` will be incremented by 1 (No matter whether the Seqnum(W) and CRC are matched in package data).

I2C Communication (HA)

<2> Reading the Normal Object

7.3 Reading From the Device

Two I^2C bus activities must take place to read from the device. The first activity is an I^2C write to set the address pointer (LSByte then MSByte). The second activity is the actual I^2C read to receive the data. The address pointer returns to its starting value when the read cycle NACK or STOP is detected.

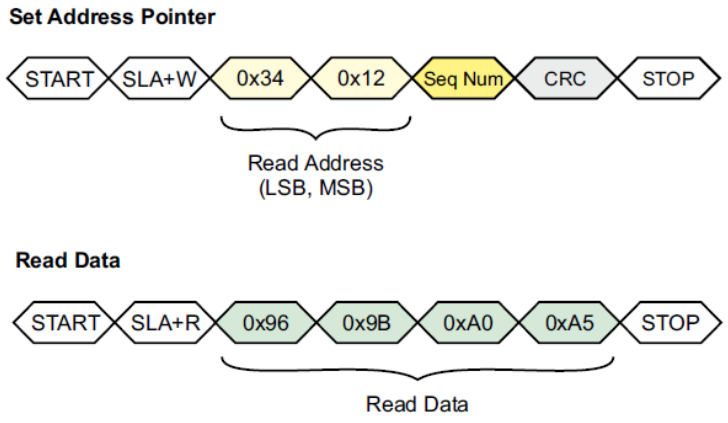
It is not necessary to set the address pointer before every read. The address pointer is updated automatically after every read operation. The address pointer will be correct if the reads occur in order. In particular, when reading multiple messages from the Message Processor T5 object, the address pointer is automatically reset to the address of the Message Processor T5 object, in order to allow continuous reads (see Section 7.3.3 “Reading Status Messages with DMA”).

The WRITE and READ cycles consist of a START condition followed by the I^2C address of the device (SLA+W or SLA+R respectively).

NOTE	Note that only certain read operations include a CRC of the data packets (see Section 7.3.1 “checksums for Read Transactions”).

Figure 7-2 shows the I^2C commands to read four bytes starting at address 0x1234.

FIGURE 7-2: EXAMPLE OF A FOUR-BYTE READ STARTING AT ADDRESS 0x1234



NOTE	At least one data byte must be read during an I^2C READ transaction; it is illegal to abort the transaction with an I^2C STOP condition without reading any data.

The object reading operation finishes in two transmit cycles, the `Seqnum(W)` and `CRC` is required in first setting address package.

<First cycle>
- The Touch controller asserting the ACK mostly means package is received and self `Seqnum(W)` will be incremented by 1 (No matter whether the Seqnum(W) and CRC are matched in package data).
- If the Seqnum(W) and CRC are both matched, slave will allocate the address pointer to current address of the package.

<Second cycle>
- The Slave will report the data without `Seqnum(W)` and `CRC` packed. Be noted only the 1st cycle is verified successfully, the data in 2nd cycle will be valid, otherwise the data is un-specified.

I2C Communication (HA)

<3> Reading message from T5 with single package

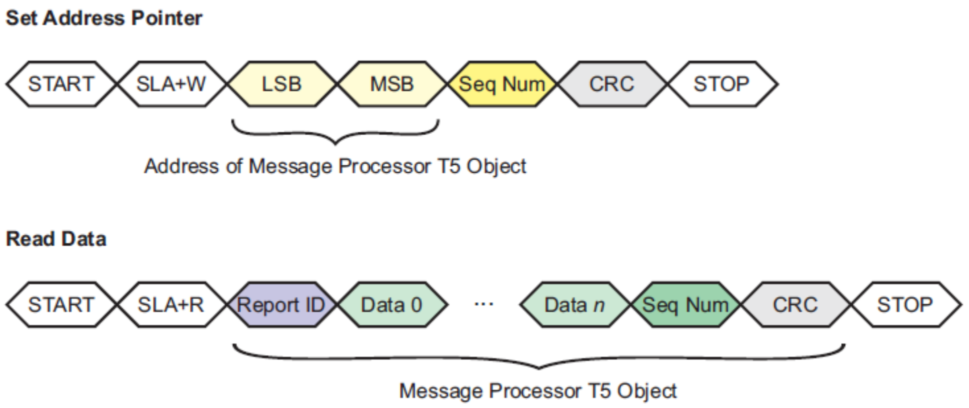
7.3.2 READING A MESSAGE FROM THE MESSAGE PROCESSOR T5 OBJECT

An I^2C read of the Message Processor T5 object contains the following bytes:

START	1 bit	I^2C START condition
SLA+R	1 byte	I^2C address of the device (see Section 7.1 “I2C Address”)
Report ID	1 byte	Message report ID
Data	1 or more bytes	The message data (size = size of Message Processor T5 MESSAGE field)
Sequence number	1 byte	The Message Processor T5 sequence number for this read. The sequence number starts at 0 for the first write after power-up/reset and is incremented by 1 for each subsequent read, wrapping round when it reaches 255.
CRC	1 byte	An 8-bit CRC for the Message Processor T5 report ID, message data and sequence number
STOP	1 bit	I^2C STOP condition

Figure 7-3 shows an example read from the Message Processor T5 object. To read multiple messages using Direct Memory Access, see Section 7.3.3 “Reading Status Messages with DMA”.

FIGURE 7-3: EXAMPLE READ FROM MESSAGE PROCESSOR T5



The object reading operation finishes in two transmit cycles, the \`Seqnum(W)\` and \`CRC\` is required in first setting address package. The 2nd response data will be packed by \`Sequm(R)\` and \`CRC\` information.

<First cycle>

- Same as before.

<Second cycle>

- The Slave will report the data with \`Seqnum(R)\` and \`CRC\` packed.

Noted only the 1st cycle is verified successfully, the data in 2nd cycle will be valid, otherwise the data is un-specified.

The \`Sequm(R)\` is different with the \`Sequm(W)\`, it’s the message’s reading cycle sequence number, and it will be omitted by host normally (useless) .

Note:

Sequm(W): The write cycle Sequm

Sequm(R): The reading cycle Sequm

- If Sequm is written without suffix () literally , we might consider it as Sequm(W) since it’s the only sequence number used in the host driver.

I2C Communication (HA)

<4> Reading message from T144 with multi-packages

7.3.3 READING STATUS MESSAGES WITH DMA

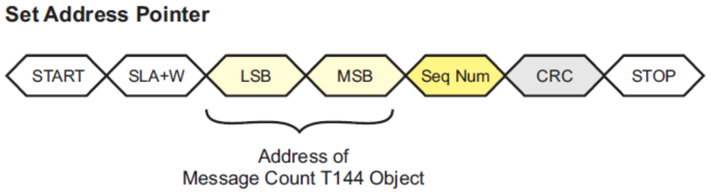
The device facilitates the easy reading of multiple messages using a single continuous read operation. This allows the host hardware to use a Direct Memory Access (DMA) controller for the fast reading of messages, as follows:

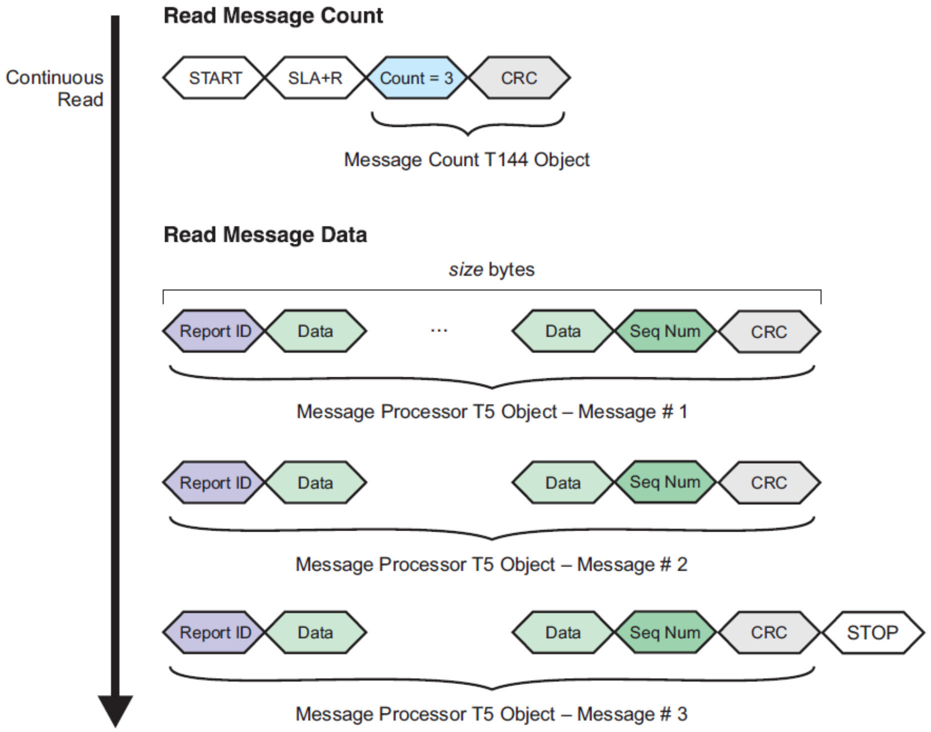
1. The host uses a write operation to set the address pointer to the start of the Message Count T144 object, if necessary. Note that the STOP condition at the end of the read resets the address pointer to its initial location, so it may already be pointing at the Message Count T144 object following a previous message read.
2. The host starts the read operation of the message by sending a START condition.
3. The host reads the Message Count T144 object (two bytes) to retrieve a count of the pending messages, plus the 8-bit CRC.
4. The host calculates the number of bytes to read by multiplying the message count by the size of the Message Processor T5 object. Note that the host should have already read the size of the Message Processor T5 object in its initialization code.

   Note that the size of the Message Processor T5 object as recorded in the Object Table includes the sequence number and checksum bytes.
5. The host reads the calculated number of message bytes. It is important that the host does *not* send a STOP condition during the message reads, as this will terminate the continuous read operation and reset the address pointer. No START and STOP conditions must be sent between the messages.
6. The host sends a STOP condition at the end of the read operation after the last message has been read. The NACK condition immediately before the STOP condition resets the address pointer to the start of the Message Count T144 object.

Figure 7-4 shows an example of using a continuous read operation to read three messages from the device.

FIGURE 7-4: CONTINUOUS READ EXAMPLE





The object reading operation finishes in two transmit cycles, the `Seqnum(W)` and `CRC` is required in first setting address package. The 2nd response data will have 2 parts:

1. T144 Object data
   - It's same as the normal `Reading the Normal Object` operation. The CRC you saw it's an object content of T144, not the packed content.
1. T5 Message data:
   - It's a bundle of messages which are same as former T5 content format of each.

Note: Normally, we will access T144 first to get out of the message count, and then using single message reading to get out the messages one by one instead of the multi-packages reading.

I2C Communication(Non-HA)

<1> Writing the Normal Object

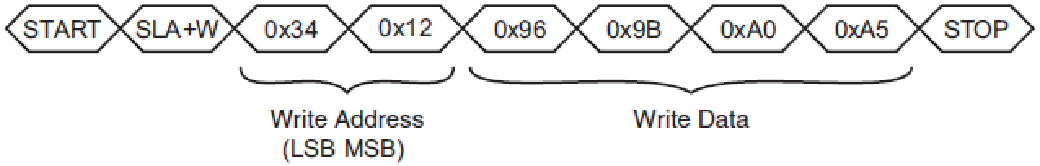
8.2 Writing To the Device

A WRITE cycle to the device consists of a START condition followed by the I^2C address of the device (SLA+W). The next two bytes are the address of the location into which the writing starts. The first byte is the Least Significant Byte (LSByte) of the address, and the second byte is the Most Significant Byte (MSByte). This address is then stored as the address pointer.

Subsequent bytes in a multi-byte transfer form the actual data. These are written to the location of the address pointer, location of the address pointer + 1, location of the address pointer + 2, and so on. The address pointer returns to its starting value when the WRITE cycle STOP condition is detected.

Figure 8-1 shows an example of writing four bytes of data to contiguous addresses starting at 0x1234.

FIGURE 8-1: EXAMPLE OF A FOUR-BYTE WRITE STARTING AT ADDRESS 0X1234

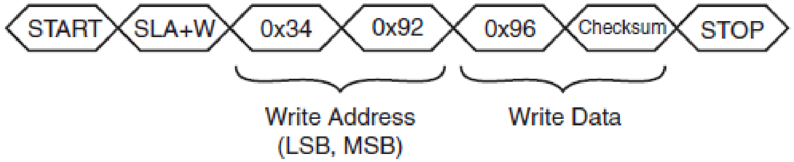


8.3 I^2C Writes in Checksum Mode

In I^2C checksum mode an 8-bit CRC is added to all I^2C writes. The CRC is sent at the end of the data write as the last byte before the STOP condition. All the bytes sent are included in the CRC, including the two address bytes. Any command or data sent to the device is processed even if the CRC fails.

To indicate that a checksum is to be sent in the write, the most significant bit of the MSByte of the address is set to 1. For example, the I^2C command shown in Figure 8-2 writes a value of 150 (0x96) to address 0x1234 with a checksum. The address is changed to 0x**9**234 to indicate checksum mode.

FIGURE 8-2: EXAMPLE OF A WRITE TO ADDRESS 0X1234 WITH A CHECKSUM



For Non-HA chips, the write action could have Checksum or not.

For Checksum mode writing, the address pointer should set to 1 at bit[8], and an extra suffix of CRC byte is added.

I2C Communication (Non-HA)

<2> Reading the Normal Object

8.4 Reading From the Device

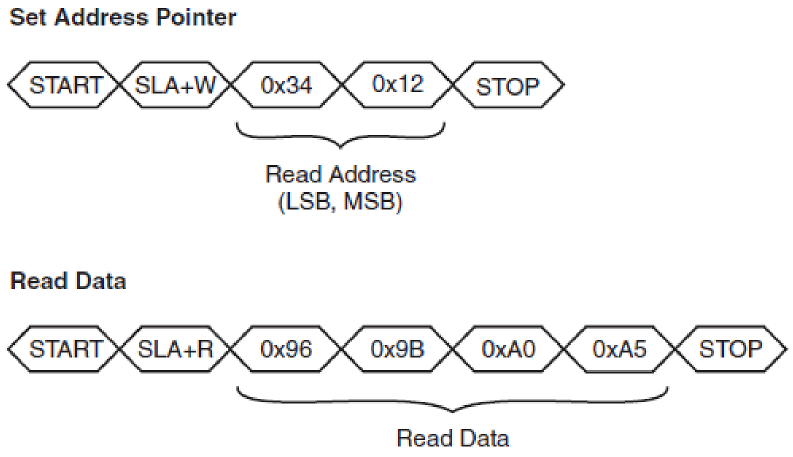
Two I^2C bus activities must take place to read from the device. The first activity is an I^2C write to set the address pointer (LSByte then MSByte). The second activity is the actual I^2C read to receive the data. The address pointer returns to its starting value when the read cycle NACK is detected.

It is not necessary to set the address pointer before every read. The address pointer is updated automatically after every read operation. The address pointer will be correct if the reads occur in order. In particular, when reading multiple messages from the Message Processor T5 object, the address pointer is automatically reset to allow continuous reads (see Section 8.5 “Reading Status Messages with DMA”).

The WRITE and READ cycles consist of a START condition followed by the I^2C address of the device (SLA+W or SLA+R respectively). Note that in this mode, calculating a checksum of the data packets is not supported.

Figure 8-3 shows the I^2C commands to read four bytes starting at address 0x1234.

FIGURE 8-3: EXAMPLE OF A FOUR-BYTE READ STARTING AT ADDRESS 0X1234

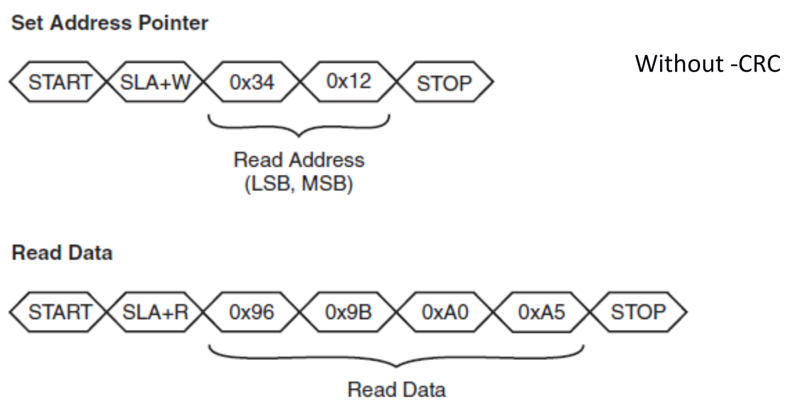


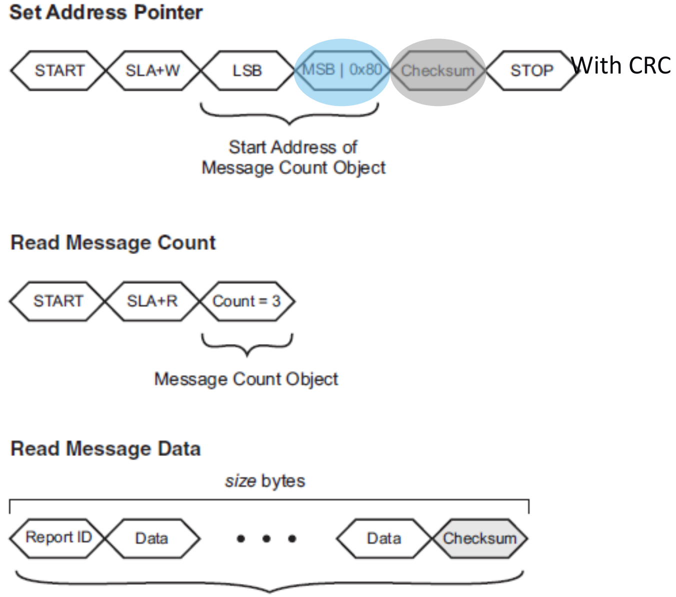
For normal object reading, the is data read without CRC is supported only.

I2C Communication (Non-HA)

<3> Reading message from T5 with single package

FIGURE 8-3: EXAMPLE OF A FOUR-BYTE READ STARTING AT ADDRESS 0X1234





The object reading of T5:

1. Non-CRC mode:
- It's same as the operation of ` Reading the Normal Object`, just set the address value to T5 address.

2. CRC mode:
- there could be CRC mode supported. The most significant bit of the MSByte of the address is set to 1. and the suffix CRC byte is added. The address value should be T5 or T44 address value.

I2C Communication (Non-HA)

<4> Reading message from T44 with multi-packages(without CRC)

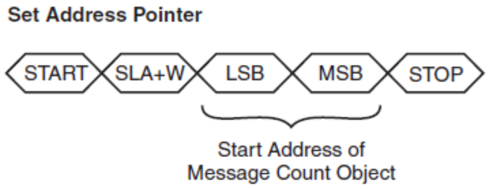
8.5 Reading Status Messages with DMA

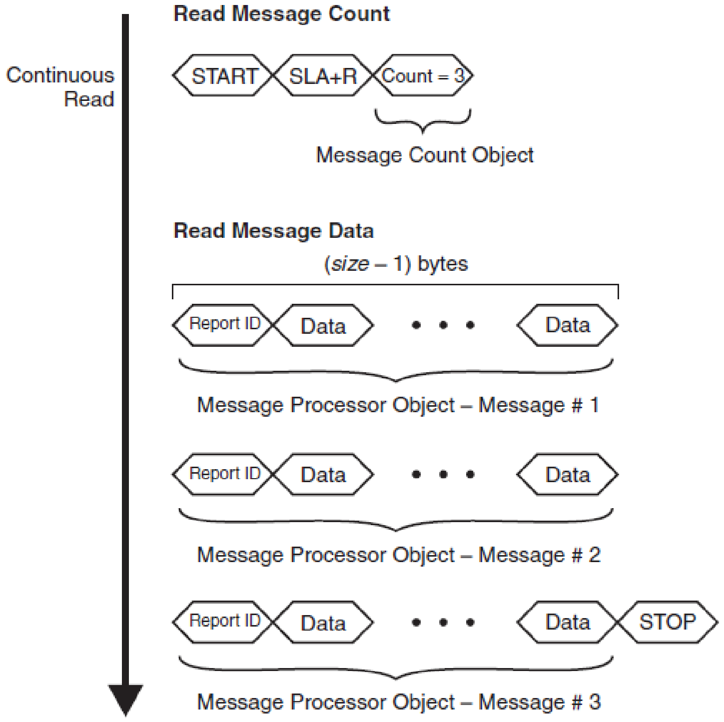
The device facilitates the easy reading of multiple messages using a single continuous read operation. This allows the host hardware to use a direct memory access (DMA) controller for the fast reading of messages, as follows:

1. The host uses a write operation to set the address pointer to the start of the Message Count T44 object, if necessary. Note that the STOP condition at the end of the read resets the address pointer to its initial location, so it may already be pointing at the Message Count T44 object following a previous message read. If a checksum is required on each message, the most significant bit of the MSByte of the read address must be set to 1.
2. The host starts the read operation of the message by sending a START condition.
3. The host reads the Message Count T44 object (one byte) to retrieve a count of the pending messages.
4. The host calculates the number of bytes to read by multiplying the message count by the size of the Message Processor T5 object. Note that the host should have already read the size of the Message Processor T5 object in its initialization code.
5. Note that the size of the Message Processor T5 object as recorded in the Object Table includes a checksum byte. If a checksum has not been requested, one byte should be deducted from the size of the object. That is: number of bytes = count × (size – 1).
6. The host reads the calculated number of message bytes. It is important that the host does *not* send a STOP condition during the message reads, as this will terminate the continuous read operation and reset the address pointer. No START and STOP conditions must be sent between the messages.
7. The host sends a STOP condition at the end of the read operation after the last message has been read. The NACK condition immediately before the STOP condition resets the address pointer to the start of the Message Count T44 object.

Figure 8-4 shows an example of using a continuous read operation to read three messages from the device without a checksum. Figure 8-5 shows the same example with a checksum.

FIGURE 8-4: CONTINUOUS MESSAGE READ EXAMPLE – NO CHECKSUM





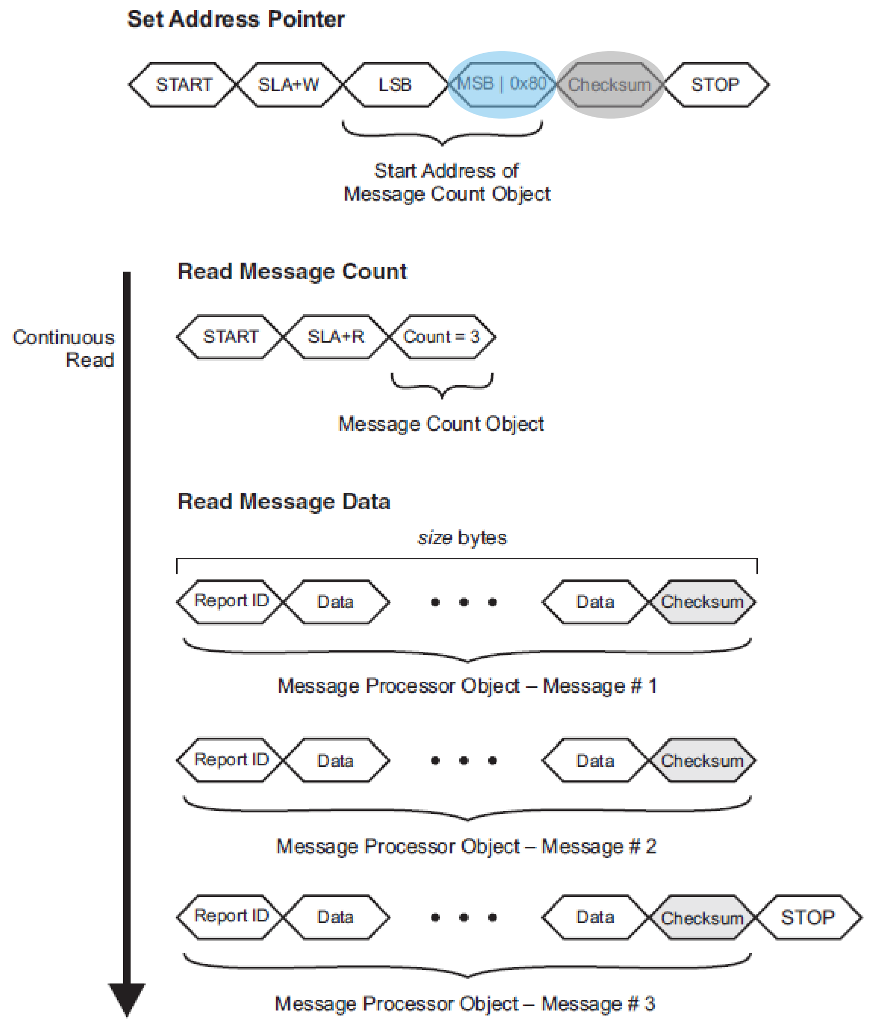
The object reading operation finishes in two transmit cycles:

1. Set the address pointer to T44 or T5
2. Data likes feedback likes `Reading the Normal Object` operation but T5 will looping to pop up when continuously reading until stop

I2C Communication (Non-HA)

<4> Reading message from T44 with multi-packages(with CRC)

FIGURE 8-5: CONTINUOUS MESSAGE READ EXAMPLE – I²C CHECKSUM MODE



CRC mode main difference with Non-CRC mode:

1. First set address pointer command should using `Writing the Normal Object` with Checksum byte.
2. The response data of T44 is without CRC byte.
3. The response data of T5 has CRC byte in each end of the package.

Interrupt assert

$\overline{\text{CHG}}$ line

The $\overline{\text{CHG}}$ line is an active-low, open-drain output that is used to alert the host that a new message is available in the Message Processor T5 object. This provides the host with an interrupt-style interface with the potential for fast response times. It reduces the need for wasteful I^2C communications.

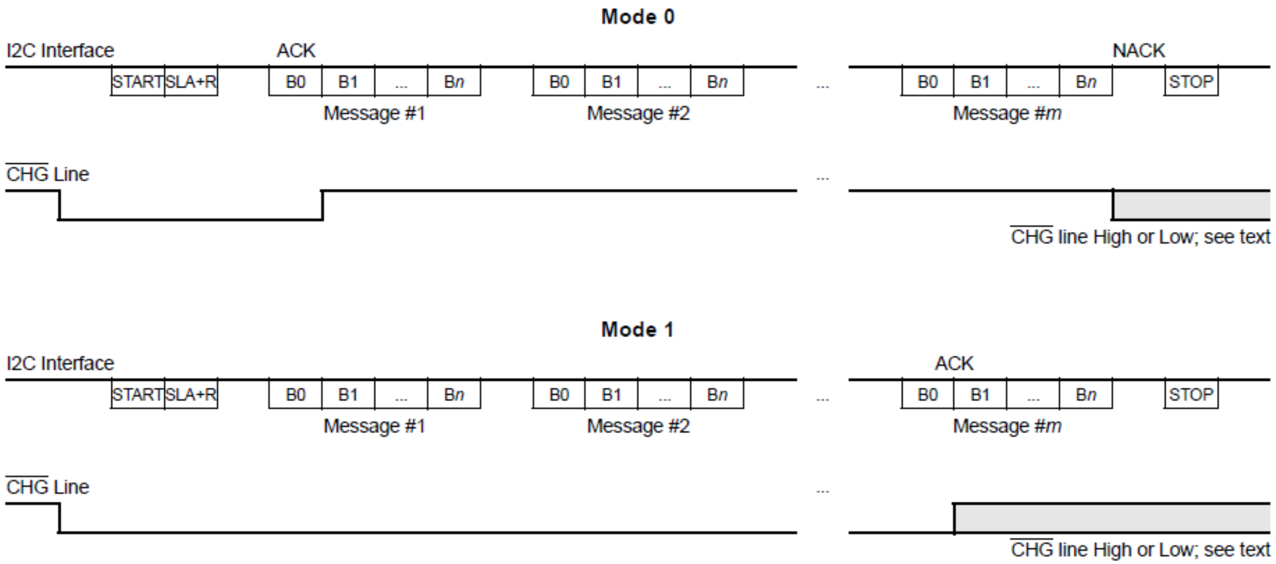
NOTE	The host should always use the $\overline{\text{CHG}}$ line as an indication that a message is ready to be read from the Message Processor T5 object; the host should never poll the device for messages.

The $\overline{\text{CHG}}$ line should always be configured as an input on the host during normal usage. This is particularly important after power-up or reset (see Section 5.0 "Power-up / Reset Requirements").

A pull-up resistor is required to VddIO (see Section 2.0 "Schematic").

The $\overline{\text{CHG}}$ line operates in two modes when it is used with I^2C communications, as defined by the Communications Configuration T18 object.

FIGURE 7-5: $\overline{\text{CHG}}$ LINE MODES FOR I^2C-COMPATIBLE TRANSFERS



Interrupt line is asserting by **low level**. That meaning if there is any message put in T5 message FIFO (or T144 is not Zero) the Interrupt is asserted.

When host acknowledge the interrupt asserting, the host will initialize the I2C transmitting. In I2C transmitting interval, the CHG could de-assert(Mode 0) and keep asserted(Mode 1), And then decide the next round level by Stop(Mode 0) or last message in T5 FIFO (Mode 1).

(Actual it's not so important to select Mode 0 or 1, we use Mode 0 as default)

For host triggering mode of interrupt handler, we should use Low level to trigger it.

But how about if host want to set **Falling Edge** to trigger interrupt handler: The chip must be configured as **Re-trigger** mode in T18 (see next page)

Note, The interrupt is asserted if any message is in T5, the message is **not only** meaningful for the touch event occurred(T15/100), **but also same** meaningful for any other object. Specially, when POR completed, there are `reset message` come out for T6. So, when chip boots up normally, the interrupt will toggle to low before your readout the message.

Interrupt assert

(Retrigen mode)

TABLE 6-19: CONFIGURATION FOR COMMUNICATIONS CONFIGURATION T18 (SPT_COMMSCONFIG_T18)

Byte	Field	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	CTRL	Reserved	RETRIGEN	Reserved		HISPEEDSPI	MODE	Reserved	
1	COMMAND	CHG line command code							

CTRL Field

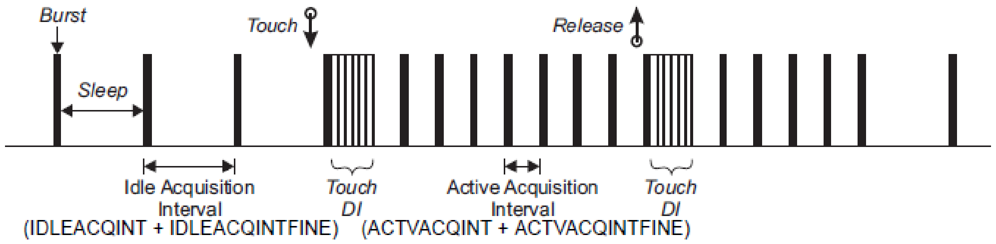
MODE: Selects the CHG line mode for I^{2}C communications over the I^{2}C interface. If this bit is set to 0 (the default), the CHG line operates in Mode 0. If this bit is set to 1, the CHG line operates in Mode 1. Refer to the relevant device *Datasheet* for more information on the CHG line modes.

HISPEEDSPI: If this bit is set to 1, debug data is sent over High Speed SPI. If this bit is set to 0 (default), debug data is sent over Normal Speed SPI (the normal speed for Microchip touchscreen products).

RETRIGEN: Enables CHG line host retriggering mode. This provides extra edge-based interrupts to the host on the CHG line. This mode is enabled if this bit is set to 1 and disabled if set to 0. If this mode is enabled, the CHG line is deasserted once every acquisition cycle for 50 µs and then re-asserted. This creates edges on the CHG line for the host in case it missed the CHG line being asserted.

This re-triggering mechanism is performed once every acquisition cycle, if the time elapsed since last time the CHG line was asserted is greater than 10 ms. This means that, if the configured acquisition interval is less than 10 ms then one or more extra acquisition cycles will be necessary to lapse for the CHG line to be re-triggered, as determined by Power Configuration T7 IDLEACQINT and ACTVACQINT (see “IDLEACQINT and ACTVACQINT Fields”).

Note that if command code 2 or 3 is set in the COMMAND field, then the command will override the RETRIGEN mode setting.



The Retrigen mode is controlled by T18 Byte[0] Bit[6].

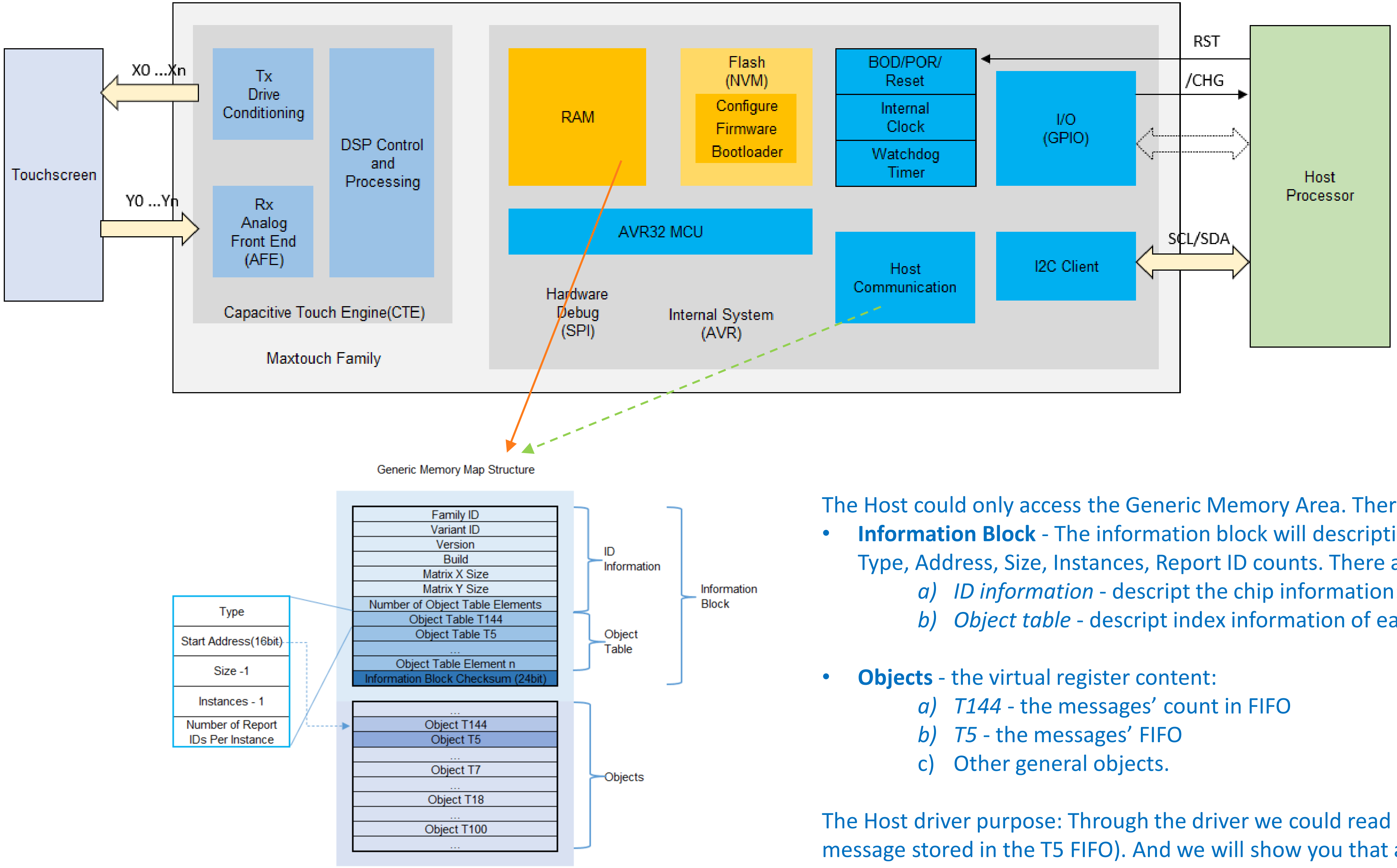
When enabled the Retrigen mode, CHG will de-assert as each acquisition cycle with 50us interval (high level pulse) .

The T7 will control the acquisition cycle, so decides the Retrigen frequency.

MICROCHIP

Driver Architecture

Driver Architecture



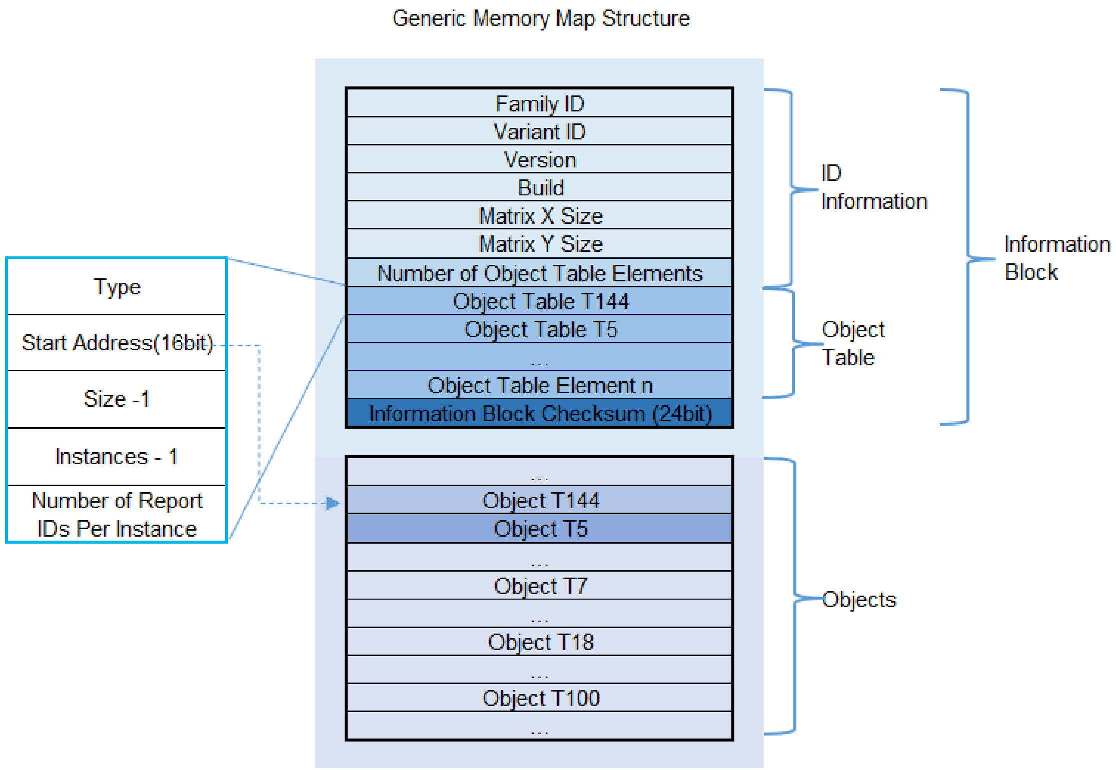
The Host could only access the Generic Memory Area. There a 2 parts of the Generic Memory:

- **Information Block** - The information block will description store the Objects description --- Type, Address, Size, Instances, Report ID counts. There are 2 sub zones:
    - *a) ID information* - descript the chip information: Chip ID, Objects count.
    - *b) Object table* - descript index information of each object.

- **Objects** - the virtual register content:
    - *a) T144* - the messages' count in FIFO
    - *b) T5* - the messages' FIFO
    - c) Other general objects.

The Host driver purpose: Through the driver we could read out finger touch event (the T100 message stored in the T5 FIFO). And we will show you that at next pages.

Driver Architecture



We have set our target: to read out finger touch event for the touch controller.
The finger event is formatted by the T100 message. And All the messages are stored in the T5 FIFO. So, to achieve the target, we need get 2 necessary details:

1. T5 Object address - We should retrieve the T100 message from there
2. T100 Report ID - The identifier of indicating which one is the T100 message in the FIFO

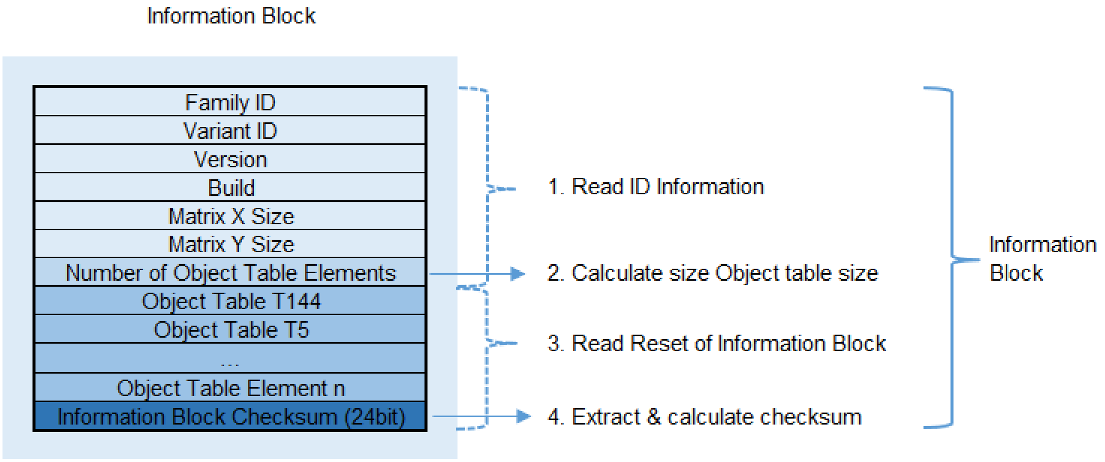
We are step by step now:

1. Read out the Information block
2. Parse the Object Table and get the T144/T5 address and T100 Report IDs
3. Interrupt Handler:
    a) Register Interrupt Function with Low Level Trigger
    b) In Interrupt handler
        - Read T144 message count
        - Read the T5 FIFO and check the Message RID. If the RID in T100 Report IDs range, then start to extract touch information

It looks easy, so it is.

Driver Architecture

(1 Read out the Information block)



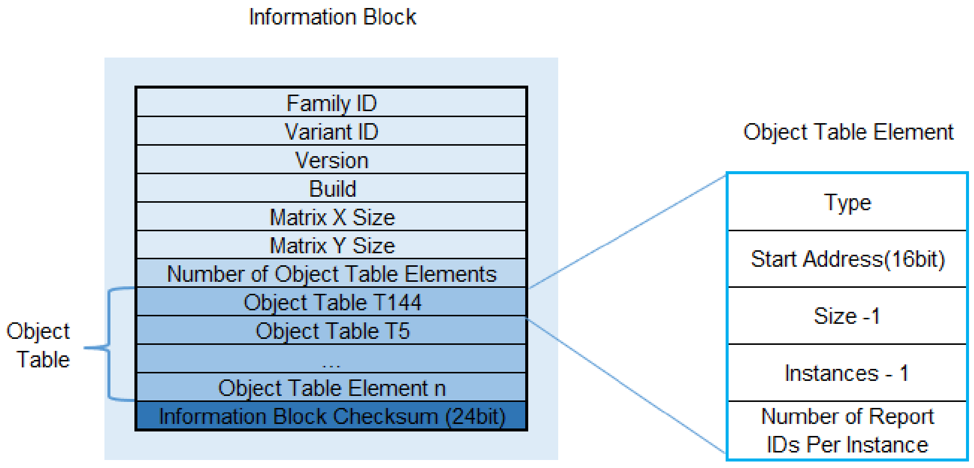

```c
/* Read 7-bytes ID information block starting at address 0 */
    size = sizeof(struct mxt_info);
    __mxt_read_reg_crc(data->client, 0, size, id_buf, data, F_R_SEQ);

/* 2 Calculate the Information block size */
    num_objects = ((struct mxt_info *)id_buf)->object_num;
    size += (num_objects * sizeof(struct mxt_object))
        + MXT_INFO_CHECKSUM_SIZE;

/* 3 Read rest of info block after id block */
    __mxt_read_reg_crc(client, MXT_OBJECT_START, (size - MXT_OBJECT_START),
        (id_buf + MXT_OBJECT_START), data, F_R_SEQ);

/* 4 Extract & calculate checksum */
    crc_ptr = id_buf + size - MXT_INFO_CHECKSUM_SIZE;
    info_crc = crc_ptr[0] | (crc_ptr[1] << 8) | (crc_ptr[2] << 16);
    calculated_crc = mxt_calculate_crc(id_buf, 0,
                       size - MXT_INFO_CHECKSUM_SIZE);
    if (info_crc != 0 && info_crc == calculated_crc) {
        /* Oh ye, CRC matched! To do next step:
           Parse Object table */
    }
```

Read out the Information block

1. Read 7-bytes ID information block starting at address 0
2. Calculate the Information block size: EachElementSize * NumObjects + CrcSize
3. Read rest of info block after id block with offset 7
4. Extract & calculate checksum

Driver Architecture

(2 Parse the Object Table and get the T5 address and T100 Report IDs)



Parse the Object Table

1. Valid Report IDs start counting from 1
2. Iterate each item in Object Table, record the address, size, and calculate the accumulated Report ID
3. The Report accumulated by: Last RID + NumInstance * NumIDsPerInstance

Now we have got the T5/T144/T100 information. The Initialization process is mostly finished. We will register the interrupt and handle it in next step.

```
/* Valid Report IDs start counting from 1 */
reportid = 1;
/* Iterate each table element */
for (i = 0; i < object_num; i++) {
        struct mxt_object *object = object_table + i;

        /* Accumulate the Report ID of current object */
        num_instances = mxt_obj_instances(object);
        if (object->num_report_ids) {
            min_id = reportid;
            reportid += object->num_report_ids *
                    num_instances;
            max_id = reportid - 1;
        } else {
            min_id = 0;
            max_id = 0;
        }

        /* Record the information of  target Object*/
        switch (object->type) {
        case MXT_GEN_MESSAGE_T5:
           data->T5_msg_size = mxt_obj_size(object); // CRC mode
           data->T5_address = object->start_address;
           break;
        case MXT_TOUCH_MULTITOUCHSCREEN_T100:
            data->multitouch = MXT_TOUCH_MULTITOUCHSCREEN_T100;
            data->T100_reportid_min = min_id;
            data->T100_reportid_max = max_id;
            data->T100_instances = num_instances;
            /* first two report IDs reserved */
            data->num_touchids = object->num_report_ids - MXT_RSVD_RPTIDS;
            break;
        case MXT_SPT_MESSAGECOUNT_T144:
            data->T144_address = object->start_address;
            data->msg_count_size = mxt_obj_size(object);
            break;
        }
}
```

Driver Architecture

(3.a Register Interrupt Function)

We need register the interrupt function as the platform API

Here the interrupt function is:
mxt_interrupt()

The Interrupt trigger mode is Low level:
irqflags = IRQF_TRIGGER_LOW

If we want to register the trigger mode as `IRQF_TRIGGER_FALLING`, be careful we should enable the `Retrigen` bit in the T18 config file.

It's quite easy, now we just wait for the interrupt event which trigger from touch control CHG line

```
static int mxt_acquire_irq(struct mxt_data *data)
{
    unsigned long irqflags = IRQF_TRIGGER_LOW;

    request_threaded_irq(&client->dev, client->irq,
                    NULL, mxt_interrupt, irqflags| IRQF_ONESHOT,
                    client->name, data);
}
```

Driver Architecture

(3.b Interrupt Handler)

Now we fill in the mxt_interrupt() stuffs.
We put mxt_process_messages_t44_t144() to handle the message.

1. Read T144 message count
2. Loop reading the message from T5 FIFO mxt_read_and_process_messages()
3. If the RID (message byte[0]) is in target Report IDs range, then start to extract touch information:
    - We call mxt_proc_t<n>_message() to handle the extraction

We will demonstrate extract the T<n> message in next step

```
static irqreturn_t mxt_interrupt(int irq, void *dev_id)
{
    return mxt_process_messages_t44_t144(data);
}

static irqreturn_t mxt_process_messages_t44_t144(struct mxt_data *data)
{
    /* Read only T144 message count */
    address = data->T144_address;
    mxt_read_reg_auto(data->client, address,
            data->msg_count_size, data->msg_buf, data);
    count = data->msg_buf[0];

    /* Process remaining messages if necessary */
    mxt_read_and_process_messages(data, count);
}

static int mxt_read_and_process_messages(struct mxt_data *data, u8 count)
{
    for (i=0; i < count; i++) {
        /* Read 1 message */
        mxt_read_reg_auto(data->client, data->T5_address,
            data->T5_msg_size, data->msg_buf, data);
        /* Process the message */
        mxt_proc_message(data, data->msg_buf);
    }
}

static int mxt_proc_message(struct mxt_data *data, u8 *message)
{
    u8 report_id = message[0];

    /* Read the T5 FIFO and check the Message RID.
       If the RID in T100 Report IDs range, then start to extract touch information */

    if (report_id >= data->T100_reportid_min &&
           report_id <= data->T100_reportid_max) {
        mxt_proc_t100_message(data, message);
    }
}
```

Driver Architecture

(3.b Interrupt Handler/T100 Touch Screen)

We should lookup the protocol about the T100 message format, that you could call Microchip support or download the MXTAN0213 document directly.

Now we got the T100 message format,
The 1st Report ID of T100 is Screen Status Message
The 2nd Report ID of T100 is Reserved
From 3rd Report ID to maximum of this instance, that's the finger tracking ID, from 1 to maximum.

So we will ignore 1st and 2nd report ID, directly decode from 3rd IDs.

TABLE 4-20: MESSAGE DATA FOR MULTIPLE TOUCH TOUCHSCREEN T100 (TOUCH_MULTITOUCHSCREEN_T100)

Byte	Field	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	TCHSTATUS	DETECT	TYPE			EVENT			
2	XPOS	X position LSByte							
3		X position MSByte							
4	YPOS	Y position LSByte							
5		Y position MSByte							
6 – 9	AUXDATA[]	Auxiliary data							

- Get TCHSTATUS BIT[7] to decide whether a touch detected
- Get out X/Y position
- Report the even

The message extracting is finished, all the things down.
We are just waiting for the interrupt occurred now!

```c
static void mxt_proc_t100_message(struct mxt_data *data, u8 *message)
{
    /* Determine id of finger touch only */
    id = (message[0] - data->T100_reportid_min - MXT_RSVD_RPTIDS);

     /* Skip SCRSTATUS events */
    if (id < 0)
        return;

    /* Get status and x/y information */
    status = message[1];
    x = get_unaligned_le16(&message[2]);
    y = get_unaligned_le16(&message[4]);

    /* report finger ID */
    input_mt_slot(input_dev, id);

    if (status & MXT_T100_DETECT) {
        /* report finger pressed */
        input_mt_report_slot_state(input_dev_sec, tool, 1);
        /* report x/y axis */
        input_report_abs(input_dev_sec, ABS_MT_POSITION_X, x);
        input_report_abs(input_dev_sec, ABS_MT_POSITION_Y, y);
    } else {
        /* report finger released */
        input_mt_report_slot_state(input_dev, 0, 0);
    }

}
```

Driver Architecture

(3.b Interrupt Handler/T15 Key Array)

We will lookup the protocol about the T15 message format, that you could call Microchip support or download the MXTAN0213 document directly.

All Keys' event in one instance share one dedicated Report ID. Each instance could support up to 16 keys. We see the format as below:

TABLE 4-7: MESSAGE DATA FOR KEY ARRAY T15 (TOUCH_KEYARRAY_T15)

Byte	Field	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	STATUS	DETECT	Reserved						
2	KEYSTATE	KEY7	KEY6	KEY5	KEY4	KEY3	KEY2	KEY1	KEY0
3		KEY15	KEY14	KEY13	KEY12	KEY11	KEY10	KEY9	KEY8

STATUS Field

Reports the current status of the object.

DETECT: Set if any key is in a touched state.

KEYSTATE Field

Reports the state of each key, one bit per key; 0 = key is untouched, 1 = key is touched.

- Byte[1]: the whole status of any key is detected.
- Byte[2~3]: indicated which key is pressed.

```
static void mxt_proc_t15_messages(struct mxt_data *data, u8 *msg)
{
    static unsigned long t15_keystatus = 0; // Key state recorder
    const unsigned int t15_num_keys = 3 ; // Number of key used

    unsigned long keystates = (msg[2] | (u16)msg[3] << 8); // Key state message

    for (key = 0; key < t15_num_keys; key++) {
        curr_state = test_bit(key, &t15_keystatus);
        new_state = test_bit(key, &keystates);

        if (!curr_state && new_state) {
           // Pressed
            __set_bit(key, &t15_keystatus);
            input_event(input_dev, EV_KEY,
                    key, 1);
        } else if (curr_state && !new_state) {
            // Released
            __clear_bit(key, &t15_keystatus);
            input_event(input_dev, EV_KEY,
                    key, 0);
        }
    }
}
```

Driver Architecture

(3.b Interrupt Handler/T9 Touch Screen)

T9 is the legacy touch screen object used with special purpose(Before S series, MPTT), we might use it in some condition and wrote the code here

Now we got the T9 message format,
Each Report ID means a finger tracking ID.

Table 5-6. Message Data for TOUCH_MULTITOUCHSCREEN_T9

Byte	Field	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	STATUS	DETECT	PRESS	RELEASE	MOVE	VECTOR	AMP	SUPPRESS	UNGRIP
2	XPOSMSB	X position MSByte							
3	YPOSMSB	Y position MSByte							
4	XYPOSLSB	X position lsbits				Y position lsbits			
5	TCHAREA	Size of touch							
6	TCHAMPLITUDE	Touch amplitude (sum of measured deltas)							
7	TCHVECTOR	Component 1				Component 2			

Note: The format for the XYPOSLSB fields depend on the resolution (10-bit or 12-bit); see page 34.

- Get STATUS BIT[7] to decide whether a touch detected
- Get out X/Y position
- Report the even

Note for different resolution of X/Y configured, the X/Y position may use different LSB decode:

```
static void mxt_proc_t9_message(struct mxt_data *data, u8 *message)
{
    id = message[0] - data->T9_reportid_min;
    status = message[1];
    x = ((u16) message[2] << 4) | ((message[4] >> 4) & 0xf);
    y = ((u16) message[3] << 4) | ((message[4] & 0xf));

    /* Handle 10/12 bit switching */
    if (data->max_x < 1024)
        x >>= 2;
    if (data->max_y < 1024)
        y >>= 2;

    input_mt_slot(input_dev, id);

    if (status & MXT_T9_DETECT) {
        /* Touch active */
        input_mt_report_slot_state(input_dev, tool, 1);
        input_report_abs(input_dev, ABS_MT_POSITION_X, x);
        input_report_abs(input_dev, ABS_MT_POSITION_Y, y);
    } else {
        /* Touch no longer active, close out slot */
        input_mt_report_slot_state(input_dev, MT_TOOL_FINGER, 0);
    }
}
```

Table 5-7. X Position Formats

XPOSMSB								XYPOSLSB							
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
10-bit Format															
512	256	128	64	32	16	8	4	2	1	N/A		Y position lsbits			
12-bit Format															
2048	1024	512	256	128	64	32	16	8	4	2	1	Y position lsbits			

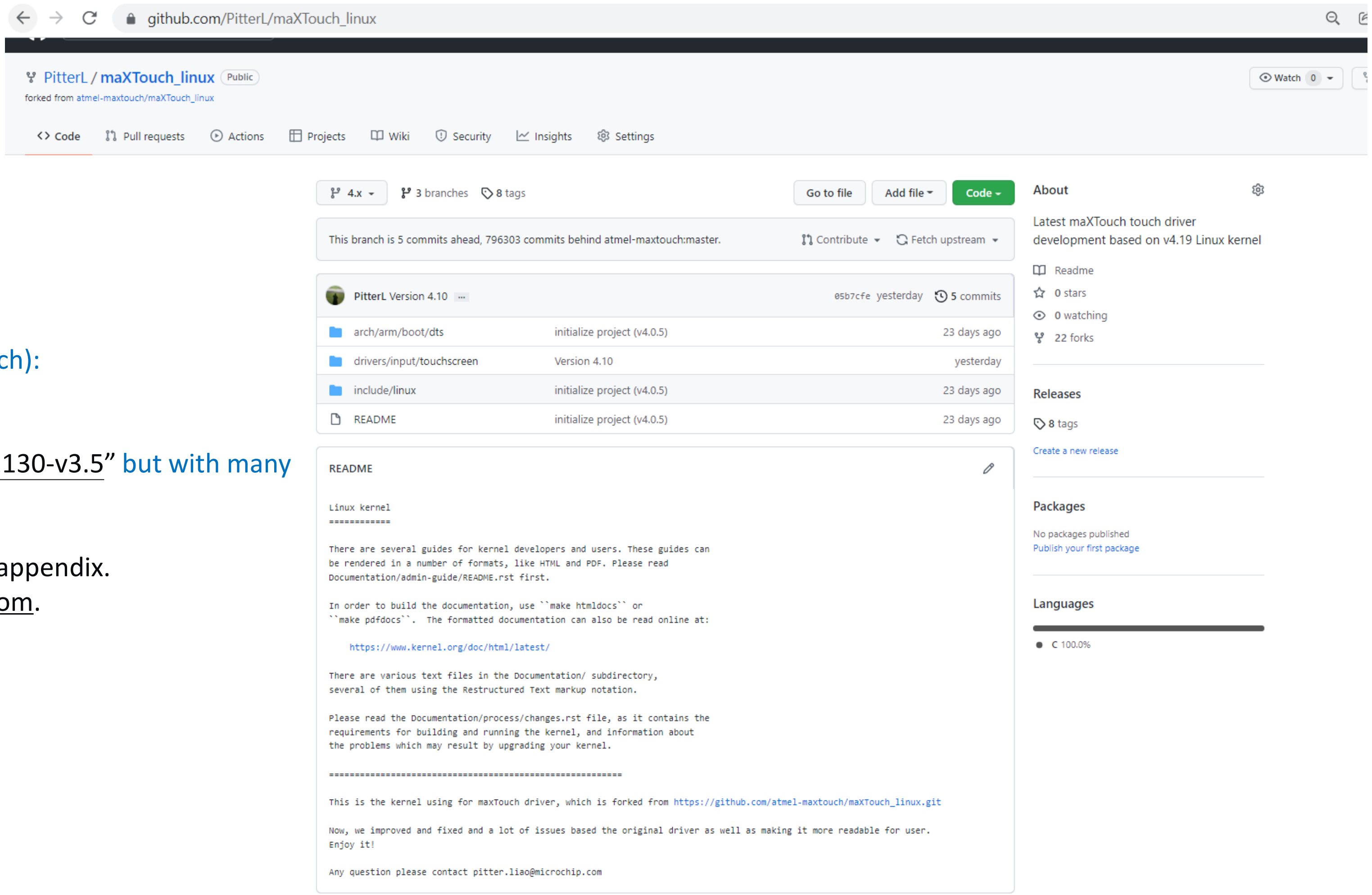
Table 5-8. Y Position Formats

YPOSMSB								XYPOSLSB							
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
10-bit Format															
512	256	128	64	32	16	8	4	X position lsbits				2	1	N/A	
12-bit Format															
2048	1024	512	256	128	64	32	16	X position lsbits				8	4	2	1

Driver Porting

Driver Porting

(Code)



You could get the driver code from the Github (4.x branch):
https://github.com/PitterL/maXTouch_linux

It's forked from official release "maXTouch-v4.19-20211130-v3.5" but with many fixed code and extensions.

You could find the diff information from this document appendix.
You could contact me with mail pitter.liao@microchip.com.

Driver Porting

(DTS)

For driver porting, we need consider below things:

1. I2C device address
2. RST/INT GPIO configure
3. Interrupt Number
4. Touch buttons keymap (optional)

The most information will refer to the DTS(Linux device tree):

We could see the reference DTS of the touch controller, which registered under the I2C bus, with the device address 0x4a.

```
# DTS setting for Hardware information

i2c1: i2c@f0018000 {
		atmel_mxt_ts@4a {
			compatible = "atmel,atmel_mxt_ts";
			reg = <0x4a>;
			reset-gpios = <&pioE 8 GPIO_ACTIVE_LOW>;
			interrupt-parent = <&pioE>;
			interrupts = <7 0x8>; /* Low level trigger */
			pinctrl-names = "default";
			pinctrl-0 = <&pinctrl_mxt_irq1 &pinctrl_mxt_rst>;
			/* t15-keymap = <139, 172, 58> */
		};
};
```

[I2C device address]

- I2C device address is hard coded in the controller (0x4A) or selected by ADDSEL PIN of chip controller(Please see schematic and chip datasheet) with different address value.

[Reset\ pin]

- The Reset\ Pin is low asserted which has 2 purposes:
  1. Cooperated with Power circuit to offer the correct POR sequence (see Chapter of Hardware specification)
  2. For Resync purpose of HA chips.

[Interrupt]

- There is interrupt number and flags. The Interrupt number is the CHG Pin mapped IRQ number in the Soc, you should query the Soc document. And please not for interrupt trigger mode, we support 2 types:
  1. IRQF_TRIGGER_LOW: the default suggested mode
  2. IRQF_TRIGGER_FALLING: be aligned with T18 of retrigen mode

[Touch buttons keymap]

For buttons support, we just set the keymap list here by:

```
t15-keymap = <139, 172, 58, 0, 22, 26, 37>
```

How many keys you have, how many key values you will set here.

- Experimental for multi-instances supported , if the chip have multi-instances of T15, the key values will be all set here. When the key value of first instance is exhausted (include the unused key nodes), left values will be for the 2nd instance. E.g. :
  - Instance<0>: 2 * 2 key array, but only 3 keys are valid, there should be 4 key values put, even if the 4th is not used. (the `0` is unused above).
  - Instance<1>: 1 * 3 keys, the first key of this instance will use the 5th key value.

Driver Porting

(I2C device address)

There are fixed I2C address or selectable address at different chips, please check the chip specification and schematic to decide what the I2C address used:

- For Fixed I2C address, it's always using 0x4A by hard coded.
- For selectable I2C address, it's controlled by the ADDSEL pin, most of chips have 2 selectable addresses, and some got even 4 addresses.

7.1 Host Communication Mode Selection – COMMSEL Pin

The selection of the host I^2C or SPI interface is determined by connecting the COMMSEL pin according to Table 7-1.

TABLE 7-1: HOST INTERFACE SELECTION

COMMSEL	Interface Selected
Connected to GND	SPI
Pulled up to VddIO (1)	I^2C

Note 1: Requires a pull-up resistor; see Section 2.0 "Schematic"

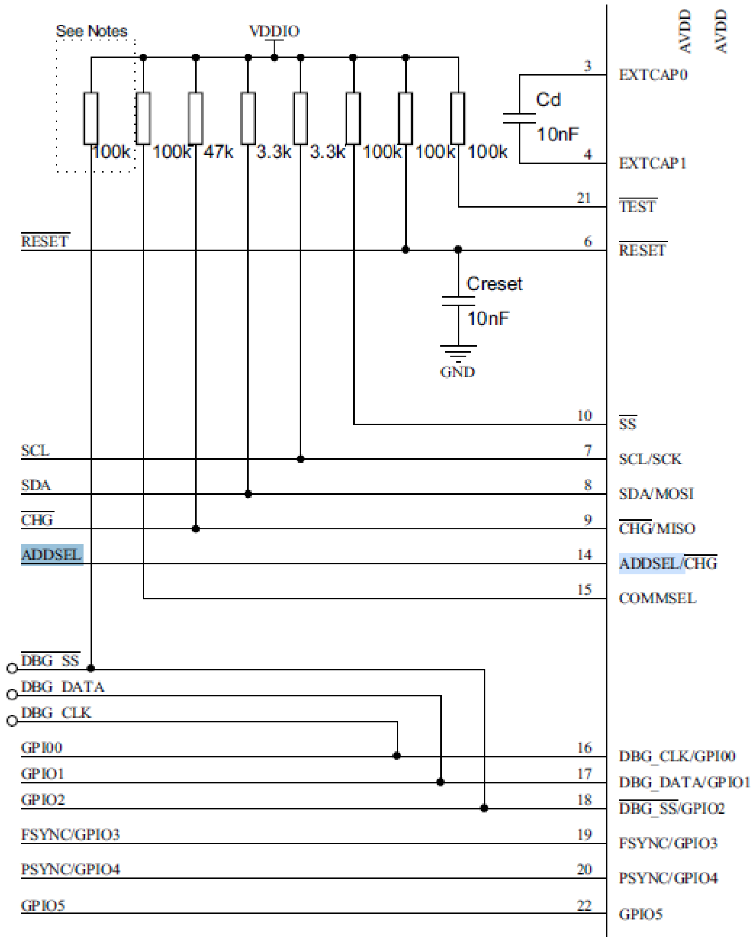
7.2 I^2C Address Selection – ADDSEL Pin

The I^2C address is selected by connecting the ADDSEL pin according to Table 7-2.

TABLE 7-2: I^2C ADDRESS SELECTION

ADDSEL	I^2C Address
Connected to GND	0x4A
Pulled up to VddIO (1)	0x4B

Note 1: Requires a pull-up resistor; see Section 2.0 "Schematic"



Driver Porting

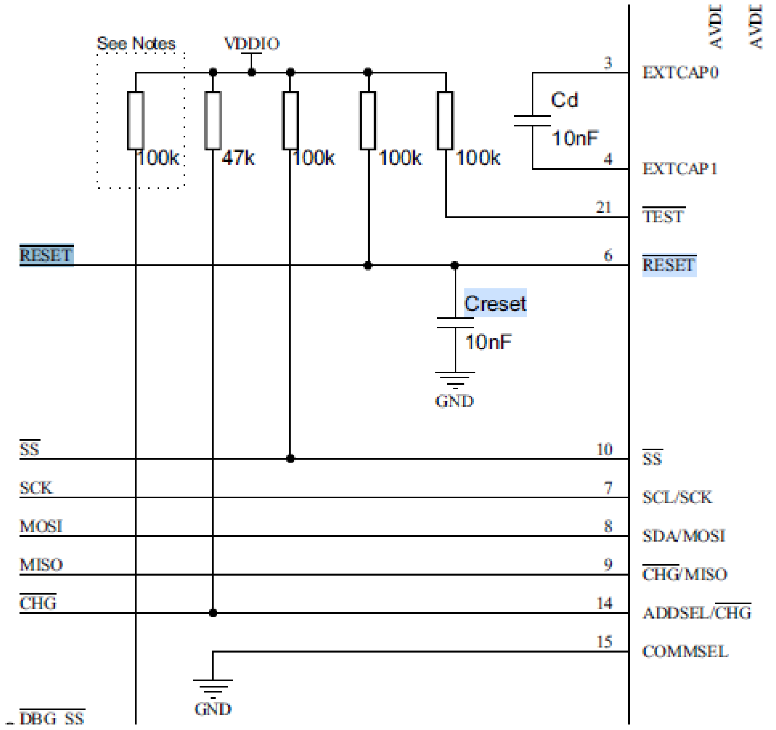
(Reset\ Pin)

The Reset\ Pin is fully controlled by the Host, it's important when Chip POR or asset Hardware reset. You have known the reset purpose as 2 points below:

1. Cooperated with Power circuit to offer the correct POR sequence (see Chapter of Hardware specification)
2. For Resync purpose of HA chips.

- For the POR, it's import to cooperated with The power circuit(Not the Reset\ Pin only) to match the POR sequence, that meaning you should set the Reset\ Pin to Low level before Power supplier, then Power supplier later. And when Power reached target voltage(~3.3v), you could release the Reset\ Pin and make it to high level.
- Power up first and assert Reset\ Pin later is meaningless for the POR sequence (this is the Hardware reset but not POR reset).

- For non-HA chips, we don't need the Resync mechanism so we may consider to leave the Reset un-connected to Host and using RC circuit to accomplish the POR sequence. This could design much more simple circuit in schematic design.
- For HA chips, Reset\ Pin is powerful for Resync the Seqnum.



Driver Porting

(Reset types)

- There are 3 types of reset: POR, Hardware and Software. The POR and Hardware reset is asserted by Reset\ Pin, Software reset is asserted by I2C command.
- The three reset mode may take different interval from dozens of milliseconds to hundreds of milliseconds by different chips.
- The POR Reset is **necessary** for chip working normally, which couldn't be replaced by Hardware or Software reset. And Hardware/Software resets are optional.
- The Reset\ Pin is asserted when voltage less than 0.3*VDDIO and de-assert when voltage is more than 0.7*VDDIO. There is Pullup resistor(~10K ohm) to VDDIO in chip internal. When VDD is less than VDDCORE(~1v). The chip's BOD is functional to block un-intended bootup.
- The Hardware reset is benefit for the Resync mechanism.

11.8 Input/Output Characteristics

Parameter	Description	Min	Typ	Max	Units	Notes
Input (All input pins connected to the VddIO power rail)						
Vil	Low input logic level	–0.3	–	0.3 × VddIO	V	VddIO = 1.8 V to Vdd
Vih	High input logic level	0.7 × VddIO	–	VddIO	V	VddIO = 1.8 V to Vdd
Iil	Input leakage current	–	–	1	μA	
RESET	Internal pull-up resistor	9	10	16	kΩ	
GPIOs	Internal pull-up/pull-down resistor					

13.6.4 RESET TIMINGS

Parameter	Min	Typ	Max	Units	Notes
Power on to $\overline{CHG}$ line low	–	48	–	ms	Vdd supply for POR VddIO supply for external reset
Hardware reset to $\overline{CHG}$ line low	–	52	–	ms	
Software reset to $\overline{CHG}$ line low	–	70	–	ms	

Note 1: Any $\overline{CHG}$ line activity before the power-on or reset period has expired should be ignored by the host. Operation of this signal cannot be guaranteed before the power-on/reset periods have expired.

Driver Porting

(Reset code)

- POR reset is achieved by Bootloader of lower level hardware. We don't have any code to show, but your thought it must match the POR sequence.

- Hardware reset is coded as below:

```
static int __hard_reset(struct mxt_data *data, u8 flag)
{
    gpio_set_value(data->reset_gpio, 0);    //Reset active
    msleep(MXT_RESET_GPIO_TIME);

    /* After reset, need to update seq num to ZERO (HA chip only) */
    mxt_update_seq_num_lock(data, true, 0x00);

    gpio_set_value(data->reset_gpio, 1);    //Reset inactive

    // Wait for Reset completed by timeout
    if (flag & F_R_WAIT) {
        msleep(MXT_RESET_INVALID_CHG);
    }

    return 0;
}
```

- Software reset is coded as below:

```
static int __soft_reset(struct mxt_data *data, u8 flag)
{
    ret = mxt_t6_command(data, MXT_COMMAND_RESET, MXT_RESET_VALUE, false);
    if (ret) {
        return ret;
    } else {
        /* After reset, need to update seq num to ZERO  (HA chip only) */
        // FIXME: there may be the thread sychronization issue for Seq num, unless it's
called under irq disabled wrapped
        mxt_update_seq_num_lock(data, true, 0x00);
    }

    /* Ignore CHG line after reset */
    if (flag & F_R_WAIT) {
        msleep(MXT_RESET_INVALID_CHG);
    }

    return 0;
}
```

Driver Porting

(Interrupt)

Normally you will get the interrupt number and irqflags from DTS in standard platform, then the interrupt number is stored in `client->irq`.

Or you may map the INT\ Pin to a valid interrupt number by the BSP API likes:

`data->irq = gpio_to_irq(The irq gpio)`

For the irqflags, if you don't use the setting in DTS, you could assign it directly by `IRQF_TRIGGER_LOW` or `IRQF_TRIGGER_FALLING`The irqflags should be matched with T18 setting Retrigen setting in CTRL byte.

```
static int mxt_acquire_irq(struct mxt_data *data)
{
    unsigned long irqflags = IRQF_TRIGGER_LOW;

#if (LINUX_VERSION_CODE >= KERNEL_VERSION(4, 0, 0))
    irqflags = 0; // Use irqd_get_trigger_type() to acquire the DTS setting automatically,
otherwise you can ommit and hard coded it.
#endif

    devm_request_threaded_irq(&client->dev, client->irq,
            NULL, mxt_interrupt, irqflags| IRQF_ONESHOT,
            client->name, data);

    return 0;
}
```

We expect the Host controller could support low level trigger by default. If not, we should enable the T18 retrigen bit and register the interrupt as falling edge.

If irqflags is set as falling edge, but the retrigen bit is not set, we may lose the interrupt sometimes, so we should check the mode by `mxt_check_retrigen()` call. In function we check the irqflags, if it's not registered as low level trigger mode, we should enable the T18 retrigen bit and write back.

```
static int mxt_check_retrigen(struct mxt_data *data)
{
    irqflags = irq_get_trigger_type(data->irq);
    if (irqflags & IRQF_TRIGGER_LOW) {
        dev_info(&client->dev, "Level triggered\n");
        return 0;
    } else {
        dev_info(&client->dev, "Get Irqflags 0x%lx, will check Retrigen mode\n", irqflags);
    }

    if (data->T18_address) {
        error = mxt_read_reg_auto(client,
            data->T18_address + MXT_COMMS_CTRL,
            1, &val, data);
        if (error)
            return error;

        if (val & MXT_COMMS_RETRIGEN) {
            dev_info(&client->dev, "RETRIGEN enabled\n");
            return 0;
        }
    }

    dev_info(&client->dev, "Enabling RETRIGEN feature\n");

    buff = val | MXT_COMMS_RETRIGEN;
    error = mxt_write_reg_auto(client,
            data->T18_address + MXT_COMMS_CTRL,
            1, &buff, data);
    if (error)
       return error;

    return 0;
}
```

Driver Porting

(Input device)

We use `mxt_initialize_input_device()` to register system input device:

- Driver will read basic information from T9 or T100 to get out resolution:
- `mxt_read_t9_resolution()`
- `mxt_read_t100_config()`

Then, call `input_allocate_device()` to allocate the input resource, and set flgas:

- EV_KEY
- ABS_X
- ABS_Y
- ABS_PRESSURE
- ABS_MT_POSITION_X
- ABS_MT_POSITION_Y
- ABS_MT_PRESSURE

If the ACPI is enabled, we will register the open and close interface.

If there are buttons in screen, we will register extra keymap value to system.
Note the keymap is set in DTS as like: `` `t15-keymap = <v1, v2, v3 …>` ``,
Or you could assign your value directly to `t15_keymap` and `t15_num_keys`.
For T15 multi-instances supported (experimental), the 2nd instance keymap will be used follow the nodes of 1st instance(even it's not used).

Finally, to register the device by call `input_register_device()`

```
static struct input_dev * mxt_initialize_input_device(struct mxt_data *data, bool primary)
{
    switch (data->multitouch) {
    case MXT_TOUCH_MULTI_T9:
           mxt_read_t9_resolution(data);

    case MXT_TOUCH_MULTITOUCHSCREEN_T100:
           mxt_read_t100_config(data, 1);
    }

    /* Register input device */
    input_dev = input_allocate_device();
    input_dev->name = primary ?
           "Atmel maXTouch Touchscreen" : "maXTouch Secondary Touchscreen";

#if (LINUX_VERSION_CODE < KERNEL_VERSION(4, 0, 0))
    set_bit(EV_ABS, input_dev->evbit);
#endif
#ifdef CONFIG_ACPI
    input_dev->open = mxt_input_open;
    input_dev->close = mxt_input_close;
#endif

    mt_flags |= INPUT_MT_DIRECT;
    input_mt_init_slots(input_dev, num_mt_slots, mt_flags);

#ifdef CONFIG_INPUT_DEVICE2_SINGLE_TOUCH
    // For single touch //
    input_set_capability(input_dev, EV_KEY, BTN_TOUCH);
    input_set_abs_params(input_dev, ABS_X, 0, data->max_x, 0, 0);
    input_set_abs_params(input_dev, ABS_Y, 0, data->max_y, 0, 0);
#endif
    input_set_abs_params(input_dev, ABS_PRESSURE, 0, 255, 0, 0);
    input_set_abs_params(input_dev, ABS_MT_POSITION_X, 0, data->max_x, 0, 0);
    input_set_abs_params(input_dev, ABS_MT_POSITION_Y, 0, data->max_y, 0, 0);

    /* For T15 Key Array */
    for (i = 0; i < data->t15_num_keys; i++) {
        input_set_capability(input_dev, EV_KEY, data->t15_keymap[i]);
    }

    input_register_device(input_dev);
}
```

Interface And Debugging

Interface And Debugging

(Sys interfaces)

There are several debug interfaces used for up layer APP:

[fw_version]
The chip Firmware version [R]
[hw_version]
The Family/variant ID [R]
[tx_seq_num]
The Seqnum driver hold currently, Or you could assign a new Seqnum to the driver [RW]
[object]
Show object table [R]
[config_crc]
Show chip config CRC value [R]
[config_ver]
Show the chip config extra information(T38 user bytes) [R]
[update_cfg]
Assign a config file name used for config updating of the chip, the config file should have been placed at the system dedicated firmware directory. [W]
[update_fw]
Assign a firmware file name used for firmware updating of the chip, the fw file should have been placed at the system dedicated firmware directory. [W]
[debug_enable]
Enable T5 message output to OS debugging console(used for `printk`). [RW]
[debug_v2_enable]
Enable T5 messaage output to `debug_msg` interface(used for uplayer App). [RW]
[debug_notify]
Watch point for `debug_msg` notification. [R]
[debug_irq]
Set the IRQ enabled or disabled, or show the current IRQ status. [RW]

- > 1 [R] IRQ is enabled more times. [W] Force IRQ enabled.
- = 1 [R] IRQ is enabled. [W] Enable IRQ if it's disabled.
- = 0 [R] IRQ is disabled. [W] Disable IRQ if it's enabled.
- < 0 [R] IRQ is disabled more times. [W] Force IRQ disabled.

```
static DEVICE_ATTR(fw_version, S_IRUGO, mxt_fw_version_show, NULL);
static DEVICE_ATTR(hw_version, S_IRUGO, mxt_hw_version_show, NULL);
static DEVICE_ATTR(tx_seq_num, S_IWUSR | S_IRUSR, mxt_tx_seq_number_show,
                   mxt_tx_seq_number_store);
static DEVICE_ATTR(object, S_IRUGO, mxt_object_show, NULL);
static DEVICE_ATTR(update_cfg, S_IWUSR, NULL, mxt_update_cfg_store);
static DEVICE_ATTR(config_crc, S_IRUGO, mxt_config_crc_show, NULL);
static DEVICE_ATTR(config_ver, S_IRUGO, mxt_cfg_version_show, NULL);
static DEVICE_ATTR(update_fw, S_IWUSR, NULL, mxt_update_fw_store);
static DEVICE_ATTR(debug_enable, S_IWUSR | S_IRUSR, mxt_debug_enable_show,
                   mxt_debug_enable_store);
static DEVICE_ATTR(debug_v2_enable, S_IWUSR | S_IRUSR, mxt_debug_v2_enable_show,
                   mxt_debug_v2_enable_store);
static DEVICE_ATTR(debug_notify, S_IRUGO, mxt_debug_notify_show, NULL);
static DEVICE_ATTR(debug_irq, S_IWUSR | S_IRUSR, mxt_debug_irq_show,
                   mxt_debug_irq_store);
static DEVICE_ATTR(crc_enabled, S_IRUGO, mxt_crc_enabled_show, NULL);
static DEVICE_ATTR(mxt_reset, S_IWUSR | S_IRUSR, mxt_reset_show, mxt_reset_store);
static DEVICE_ATTR(selftest, S_IWUSR | S_IRUSR, mxt_selftest_show, mxt_selftest_store);
```

[crc_enabled]
Set the the object access with CRC enabled, or show whether the CRC accessing is enabled [RW]
[mxt_reset]
Reset the chip, or check whether chip is still executing Reset by APP (Non-App reset is un-wachable) [RW]

- 0: Nothing
- 1: Any Reset with completion waiting (equal BITS[1]||[2][3] masked)
- BIT(1): Software reset
- BIT(2): Hardware reset
- BIT(3): Reset with completion waiting

[selftest]
Execute the selftest command of T25/T10, note the selftest object should be configured preliminarily. Here we just sent test command to target register. Reading the interface will show the test result. [RW]

Interface And Debugging

(Config update)

The config update through 2 ways, the driver callback in initialization or the Sys interface.

Driver callback at in initialization:

- In mxt_initialize(), there is default calling of the config update with the dedicated name `"maxtouch.cfg"`, the interface will callback `mxt_config_cb()` when file system is mounted. The callback will call `mxt_configure_objects()` later with the actual parameter of config file handle read from the file system. And then `mxt_update_cfg()` is called and the chip will be flashed with the new config data.

Sys interface:

- We have seen there is an `update_cfg` interface in last page introduced. We could echo a config name to the interface. That trigger the `mxt_update_cfg_store()` to be called. And then, if the driver had found the target config in OS by , it will call `mxt_configure_objects()` likely first method, then do the config updated.

Note, the Hardware version and Config CRC will be checked first for the flash data. And whatever method used, the config file should have been placed at the system dedicated firmware directory of the system:
The OS will search the directory as below or by system decided:

```
// drivers/base/firmware_loader/main.c
static const char * const fw_path[] = {
    fw_path_para,
    "/lib/firmware/updates/" UTS_RELEASE,
    "/lib/firmware/updates",
    "/lib/firmware/" UTS_RELEASE,
    "/lib/firmware"
};

/*
 * Typical usage is that passing 'firmware_class.path=$CUSTOMIZED_PATH'
 * from kernel command line because firmware_class is generally built in
 * kernel instead of module.
 */
module_param_string(path, fw_path_para, sizeof(fw_path_para), 0644);
MODULE_PARM_DESC(path, "customized firmware image search path with a higher priority than
default path");
```

For pure Linux, it's `"/lib/firmware"`default, and for Android, it will be decided by system variable, mostly like `"/vendor/etc/firmware"` `"/etc/firmware"` …

```
#define MXT_CFG_NAME        "maxtouch.cfg"

static int mxt_initialize(struct mxt_data *data)
{
    if (true){
        /* As built-in driver, root filesystem may not be available yet */
        error = request_firmware_nowait(THIS_MODULE, true, MXT_CFG_NAME,
                        &client->dev, GFP_KERNEL, data,
                        mxt_config_cb);
        if (error) {
            dev_warn(&client->dev, "Failed to invoke firmware loader: %d\n",
                error);
        }
    } else {
        mxt_configure_objects(data, NULL);
    }
}
```

```
static ssize_t mxt_update_cfg_store(struct device *dev,
        struct device_attribute *attr,
        const char *buf, size_t count)
{
    mxt_update_file_name(dev, &file_name, buf, count);

    request_firmware(&cfg, file_name, dev);

    mxt_configure_objects(data, cfg);
}
```

Interface And Debugging

(Firmware update)

The firmware update is rarely used because the firmware is embedded into the chip when ship out, but you may upgrade it through the Sys interface `update_fw`:

- We have seen there is an `update_fw` interface in last page introduced. We could echo a firmware name to the interface. That trigger the `mxt_update_fw_store()` to be called. And then, if the driver had found the target FW file in OS by , it will call `mxt_load_fw()` to enter bootloader mode and update the flash data. The firmware data sent each frame by `mxt_bootloader_write()` and `mxt_bootloader_read()` api, which uses different I2C address as normal communication.

- After firmware flashed to chip, host should wait for the chip reboot to normal mode with `MXT_FW_FLASH_TIME` time interval. If the firmware data is valid (firmware data match the silicon information and CRC passed), chip will bootup into App mode, we will re-start to initialize the chip with it's new information block. And call the `mxt_config_cb()` to check whether we get new config data match this new firmware.

```
static ssize_t mxt_update_fw_store(struct device *dev,
                    struct device_attribute *attr,
                    const char *buf, size_t count)
{
    mxt_update_file_name(dev, &file_name, buf, count);

    mxt_clear_cfg(data);

    mxt_load_fw(dev, file_name);

    msleep(MXT_FW_FLASH_TIME);

    __mxt_reset(data, F_RST_ANY);   // Any reset

    /* Read infoblock to determine device type */
    mxt_read_info_block(data);

    mxt_acquire_irq(data);

    request_firmware_nowait(THIS_MODULE, true, MXT_CFG_NAME,
                    dev, GFP_KERNEL, data,
                    mxt_config_cb);
}
```

```
static int mxt_load_fw(struct device *dev, const char *fn)
{
    request_firmware(&fw, fn, dev);

    /* Check for incorrect enc file */
    mxt_check_firmware_format(dev, fw);

    mxt_t6_command(data, MXT_COMMAND_RESET, MXT_BOOT_VALUE, false);

    mxt_update_seq_num_lock(data, true, 0x00);

    msleep(MXT_RESET_TIME);

    /* Do not need to scan since we know family ID */
    mxt_lookup_bootloader_address(data, 0);

    INIT_COMPLETION(data->bl_completion);

    mxt_check_bootloader(data, MXT_WAITING_BOOTLOAD_CMD, false);

    /* Unlock bootloader */
    mxt_send_bootloader_cmd(data, true);

    while (pos < fw->size) {
        /* Write one frame to device */
        …
        mxt_bootloader_write(data, &fw->data[pos], frame_size);

        mxt_check_bootloader(data, MXT_FRAME_CRC_PASS, true);
        …
    }

    INIT_COMPLETION(data->bl_completion);

    msleep(MXT_BOOTLOADER_WAIT);    /* Wait for chip to leave bootloader*/

    ret = mxt_wait_for_completion(data, &data->bl_completion,
                    MXT_BOOTLOADER_WAIT);}
```

For the flow chat of bootloader mode, please refer the MXTAN0216_maXTouch_Bootloader document.

Interface And Debugging

(Experimental Selftest)

We have T25/T10 selftest interface of `selftest`:

- It will accept a test command to execute the test and feedback result by readout the same interface.
- The cmd input is set as below for T25(Non-HA) or T10(HA):
- We should set all the appropriate parameter in chip config before issue the test command.
- Please refer the protocol for more test details

- Command of T25 (Normal we will issue `0xFE` for the command):

TABLE 6-11: TEST COMMANDS

Command	Description
0x00	The CMD field is set to 0x00 after test completed
0x01	Test for AVdd power present
0x12	Run the pin fault test
0x17	Run the signal limit test
0xFE	Run all the tests

- Result of T25 (We will get 0xFE if passed) :

TABLE 6-15: RESULT CODES

Code	Test Result
0xFE	All tests passed.
0xFD	The test code supplied in the CMD field is not associated with a valid test.
0x01	AVdd is not present. This failure is reported to the host every 200 ms.
0x12	The test failed because of a pin fault. The INFO fields indicate the first pin fault that was detected (see Table 6-16). Note that if the initial pin fault test fails, then the Self Test T25 object will generate a message with this result code on reset.
0x17	The test failed because of a signal limit fault.

```
static ssize_t mxt_selftest_store(struct device *dev,
    struct device_attribute *attr, const char *buf, size_t count)
{
    mxt_set_selftest(data, cmd, true);
}
```

- Command of T10 (Normal we will issue `0x3E` for the command):

TABLE 6-4: TEST COMMANDS

Command	Description
0x00	No command
0x31	Run the Power test
0X32	Run the pin fault test
0x33	Run the signal limit test
0x3E	Run all the tests in the order above
All other values	Reserved; no effect

- Result of T10 (We will get 0x31 if passed):

TABLE 6-6: STATUS FIELD

Result Code	Meaning	INFO[] Field Data
0x31	All on-demand tests have passed	Not used; the INFO field consists of reserved bytes only
0x32	An on demand test has failed	See Section "INFO Field – Test Failed"
0x3F	The test code supplied in the CMD field is not associated with a valid test	See Section "INFO Field – Invalid Command Code"
0x11	All POST tests have completed successfully	Not used; the INFO field consists of reserved bytes only
0x12	A POST test has failed	See Section "INFO Field – Test Failed"
0x21	All BIST tests have completed successfully	Not used; the INFO field consists of reserved bytes only
0x22	A BIST test has failed	See Section "INFO Field – Test Failed"
0x23	BIST test cycle overrun	Not used; the INFO field consists of reserved bytes only

Resync (HA)

Resync

(Algorithm)

For HA silicon, there are extra stuffed information that named `Seqnum` in each frame (See `I2C Communication` Chapter). If the Seqnum is mis-matched between host and touch controller, the touch controller will not execute the command sent from host or can't respond correct data to host. That will damage the data communication between them.

So how could we do if Seqnum is occasional missed, can we retrieve it back specified? Unfortunately, the official document don't talk about it in details, we should achieve it in our experience.

The key point is when Seqnum is lost, mostly the CRC in each package is incorrect, then we must consider the Resync algorithm to communicate later.

We mostly consider the Resync at 2 scenarios:

- First bootup when the information block CRC verified failed.
- In interrupt processing, the CRC of T44 or T5 are incorrect.

How could we think of the Resync algorithm to make communication workable?
We should know the facts from the test result first:

- We can verify the response data by the checksum to decide whether a Seqnum is correct.
- There are 3 scenarios that data respond with checksum byte:
    1) Read Information Block
    2) T44 data
    3) T5 data
- If the Seqnum is mismatched, the address pointer of the firmware is stuck somewhere, the feedback data will be consistent each time.
- The correct Seqnum can conclude a correct data, but correct data is **not** sufficient to conclude a correct Seqnum, because the address pointer might be occasional same as you want to set this time.

With all above information, we will design the algorithm by reading Information Block with address pointer offset (twice read) to verify the result:

- Read out the whole Information block with address offset 0, then verified CRC
- Read out left pieces of Information block with offset 7, then verified the CRC again.
- If the information block verified twice, we consider it's correct Seqnum retrieved.

But you may ask why we chose the Information block instead of T44/T5 which will be shorted data transferred each time. Here we considered the T44 and T5 data is variable and can't reuse the exist ID information we may have in initialization stage (We consider the Interrupt data broken in priority). But you could think about to use T44 to speed up the Resync algorithm.

Now, we will assume the 3 scenarios to retrieve the Seqnum:

1) The chip may occasionally reset (By ESD/BOD or some other things), the Seqnum is mostly Zero and nearby.
    - We Search from with Seqnum 2 with 4 times loop. (CRC missed, and one debounce, so start from will be 2)
2) The Seqnum is somewhere unknown.
    - We search from latest known value with a step value debounce (seqnum_last + step), and `256 + step` times loop.
3) The Seqnum is ZERO by Hardware reset.
    - We search from Zero with 3 times loop

We will show it in next page.

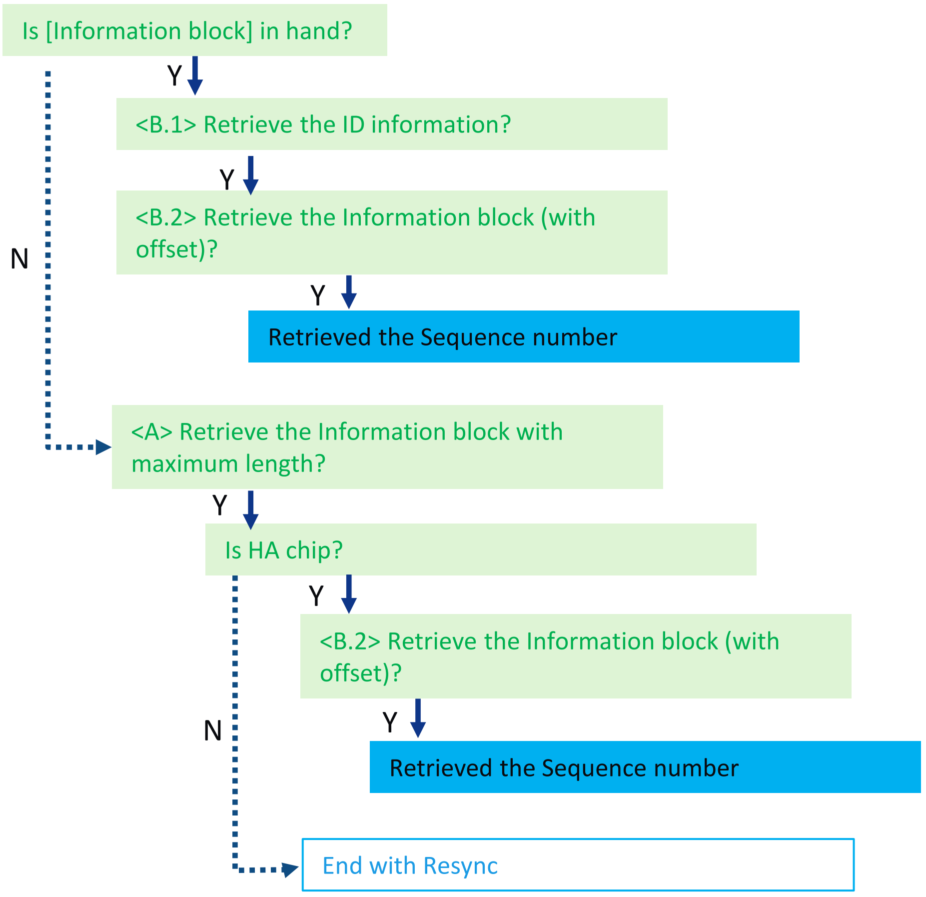
Resync

(Algorithm)

Round 0: Search from with Seqnum 2 with 4 times retry

Round 1: We search from latest known value with a step value debounce

Round 2: The Seqnum is ZERO by Hardware reset




```
static int mxt_resync_comm(struct mxt_data *data)
{
    for ( i = 0; i < 3 && synced != SYNCED_COMPLETED; i ++ ){
        if (i == 0) {
        // <Round.0>: Assumed Seqnum change to `0` with unknown reason, set to 2 (1 CRC + 1 debounce)
        }else if (i == 1) {
        // <Round.1>: use the overlfow method to retrieve the seq num, set to `seqnum_last` + step
        } else {
        // <Round.2>: use Hardware reset to retrieve the seq number, set to 0
        }

        seqnum = mxt_curr_seq_num(data);
        for ( j = 0; j < count + step && synced != SYNCED_COMPLETED; j += step ) {
            for ( k = 0; k < step; k++ ) {
                mxt_update_seq_num_lock(data, true, seqnum);  // Fix the Seq number in `K` Round
        /* If we have correct information block in hand, we can compare the buffer data with it directly */
                if (info) {
                    <Check Point B.1> Check the ID information
                    if (memcmp(dev_id_buf, info, dev_id_size)) {
                        // ID information mis-matched, read ID information first
                    } else {
                        // ID information matched, to read out all left information block at <Check Point B.2>
                    }
                } else {
                    // <Check Point A> No info block yet, to read out all the information block
                }
                // Start read operation
                __mxt_read_reg_crc(client, reg, size, buf, data, F_R_SEQ);
                if (buf == dev_id_buf && size == dev_id_size) {
                    if (info && !memcmp(dev_id_buf, info, dev_id_size)) {
          // <B.1> result is valid - ID information matched, loop again to read left information block part <B.2>
                    }
                } else {
                        calculated_crc = mxt_calculate_crc(dev_id_buf);
                        if (info_crc == calculated_crc) {
                            if (!info) {
                                // <Result A> verified, first time to readout information block <B.2>
                                if (mxt_lookup_ha_chips(dev_id_buf)) {
                                    // <A> found HA chips, save the info block valid and loop to check again
                                    info = dev_id_buf;
                                } else {
                                    synced = SYNCED_COMPLETED;  // <A> found non-HA chip, Resync is complete
                                }
                            } else {
                  // <Result B.2> verified, second time to readout information block, the Resync is completed
                            }
                        }
                    } else {
                        // <Result A or B.2> invalid: Information block CRC check failed
                    }
                }
            }
        }
    }
}
```

FAQ

FAQ

(Flowchart/initialization)

- **Is the newest code compatible with Non-HA chips:**

Yes

- **How reliable is there for the Resync algorithm:**

None could guarantee it because you don’t release know whether the Seqnum is matched , there is not indicator to tell outside. The actual use case is complex and there might be some hardware issue exist also All we know it try to retrieve it by the algorithm. We have done the tension test as descripted in the driver version log. You Could referred it.

We have deployed the driver into 5 projects until Dec 31/2021:
You are not the first mover!

Thank you!

Appendix (A)
Code comparison

Code comparison

(Flowchart/initialization)

mxt_probe()

- Driver version control
- Move hardware initialization information into mxt_initialize()
- Call mxt_remove() to release resource if failed

```
static int mxt_probe(struct i2c_client *client, const struct i2c_device_id *id)
{

    dev_info(&client->dev, "ATMEL MaXTouch Driver version %s\n", DRIVER_VERSION_NUMBER);

    error = mxt_parse_device_properties(data);
    if (error) {
        dev_err(&client->dev, "Parse device properties failed %d\n", error);
        goto failed;
    }

    error = mxt_parse_gpio_properties(data);
    if (error) {
        dev_warn(&data->client->dev, "Skipped to use hardware reset\n");
    } else {
        __mxt_reset(data, F_RST_HARD);
    }

    error = mxt_initialize(data);
    if (error) {
        dev_err(&client->dev, "mxt initialize failed %d\n", error);
        goto failed;
    }

    /* Enable debugfs */
    atmel_mxt_ts_prepare_debugfs(data, dev_driver_string(&client->dev));

    /* Removed the mxt_sys_init and mxt_debug_msg_init */
    /* out of mxt_initialize to avoid duplicate inits */

    error = mxt_sysfs_init(data);
    if (error) {
        dev_err(&client->dev, "sysfs init failed %d\n", error);
        goto failed;
    }

    error = mxt_debug_msg_init(data);
    if (error) {
        dev_err(&client->dev, "debug msg init failed %d\n", error);
        goto failed;
    }

    mutex_init(&data->debug_msg_lock);

    return 0;
failed:
    mxt_remove(client);
    return error;
}
```

Code comparison

(Flowchart/initialization)

`mxt_read_info_block()`
Retry and Re-synce called if initialization failed first time. This avoid probe driver failed.

`mxt_configure_objects()`
Free resource after config updated, the `mxt_initialize_input_device()` will initialize it with newest config setting

Call `mxt_initialize_input_device()` with actual parameter to decide the first or second input device

Call `mxt_init_t7_power_cfg()` and `mxt_check_retrigen()` here

```
static int mxt_read_info_block(struct mxt_data *data)
{
    struct i2c_client *client = data->client;
    int error = 0;
    int i, retries = 2;

    /* read info block with retries */
    for ( i = 0; i < retries; i++) {
        error = __mxt_read_info_block(data);
        if (error) {
            dev_warn(&client->dev, "Read Info block check resync %d", i + 1);

            if (mxt_resync_comm(data)) {
                // resync failed directly exit
                dev_info(&client->dev, "Read Info block resync failed, exit");
                break;
            }
        } else {
            /* successful */
            break;
        }
    }

    return error;
}
```

```
static int mxt_configure_objects(struct mxt_data *data,
                 const struct firmware *cfg)
{
    if (cfg) {
        error = mxt_update_cfg(data, cfg);
        if (error < 0) {
            dev_warn(dev, "Error %d updating config\n", error);
        } else if (error == 0) {
            dev_info(dev, "Skip update config file\n");
        } else {
            dev_info(dev, "Config file updated, release the input device\n");
            mxt_free_input_device(data);
            mxt_free_second_input_device(data);
        }
    }

    if (!data->input_dev) { // Check the major device only
        if (data->multitouch) {
            dev_info(dev, "mxt_config: Registering devices\n");
            data->input_dev = mxt_initialize_input_device(data, true);
            if (!data->input_dev) {
                dev_warn(dev, "Error to Register primary device\n");
                return error;
            }

            if (data->T100_instances > 1) {
                data->input_dev_sec = mxt_initialize_input_device(data, false);
                if (!data->input_dev_sec) {
                    dev_warn(dev, "Error ot Register secondary device\n");
                }
            }
        } else {
            dev_warn(dev, "No touch object detected\n");
        }

        // FIXME: when should call mxt_debug_deinit()
        mxt_debug_init(data);
    }

    /* T7 config may have changed */
    mxt_init_t7_power_cfg(data);

    /* check T18 retrigen bit with irqflags */
    error = mxt_check_retrigen(data);
    if (error) {
        dev_warn(dev, "RETRIGEN Not Enabled or unavailable\n");
    }
}
```

Code comparison

(Flowchart/initialization)

`mxt_parse_device_properties()`
Parse t15-keymap in default code initialization.

`mxt_parse_gpio_properties()`
Support legacy reset-gpio operation

```
static int mxt_parse_device_properties(struct mxt_data *data)
{
    struct device *dev = &data->client->dev;
    int error;

    error = __mxt_parse_device_properties(data, "gpio-keymap", &data->t19_keymap, &data->t19_num_keys);
    if (error) {
        dev_err(dev, "failed to parse gpio keymap\n");
        return error;
    }


    error = __mxt_parse_device_properties(data, "t15-keymap", &data->t15_keymap, &data->t15_num_keys);
    if (error) {
        dev_err(dev, "failed to parse t15 keymap\n");
        return error;
    }

    return 0;
}
```

```
static int mxt_parse_gpio_properties(struct mxt_data *data)
{
    struct device *dev = &data->client->dev;

#if (LINUX_VERSION_CODE >= KERNEL_VERSION(4, 0, 0))
    // Using gpiod_xxx interface
    data->reset_gpio = devm_gpiod_get_optional(dev,
                           "reset", GPIOD_OUT_LOW);
    if (IS_ERR_OR_NULL(data->reset_gpio)) {
        if (data->reset_gpio == NULL) {
            dev_warn(dev, "Warning: reset-gpios not found or undefined\n");
        } else {
            dev_err(dev, "Failed to get reset_gpios\n");
        }
        return -EPERM;
    }
    else {
        gpiod_direction_output(data->reset_gpio, 1);    /* GPIO set output */
        dev_info(dev, "Direction is ouput\n");

    }
#else
    struct device_node *np = data->client->dev.of_node;

    // Using legacy gpio_xxx interface
    data->reset_gpio = of_get_named_gpio_flags(np, "reset-gpios",
                            0, NULL);
    if (!gpio_is_valid(data->reset_gpio)) {
        if (data->reset_gpio == 0) {
            dev_warn(dev, "Warning: reset-gpios not found or undefined\n");
        } else {
            dev_err(dev, "Failed to get reset_gpios\n");
        }

        return -EPERM;
    } else {
        gpio_direction_output(data->reset_gpio, 1); /* GPIO set output */
        dev_info(dev, "Direction is ouput\n");
    }
#endif

    return 0;
}
```

Code comparison

(IRQ processing)

Use mxt_acquire_irq() to manage irq registration and irqflags mode setting. Compatible the standard kernel(DTS) or non-stand kernel (assign the irqflags)

Use atomic operation for tracking `irq_processing`
IRQ disable enable API for up layer should check ` irq_processing` status before disable or enable. It could only be set in mxt_acquire_irq() and mxt_disable_irq().

```
static ssize_t mxt_debug_irq_store(struct device *dev,
    struct device_attribute *attr, const char *buf, size_t count)
{
    struct mxt_data *data = dev_get_drvdata(dev);
    s8 i;
    ssize_t ret;

    if (kstrtos8(buf, 0, &i) == 0 && i < 2) {
        if (i > 0) {
            if (i > 1 || atomic_read(&data->irq_processing) <= 0) {
                mxt_acquire_irq(data);
            }
        } else {
            if (i < 0 || atomic_read(&data->irq_processing) > 0) {
                mxt_disable_irq(data);
            }
        }

        dev_info(dev, "%s(%d)\n", i ? "Debug IRQ enabled" : "Debug IRQ disabled", i);
        ret = count;
    } else {
        dev_dbg(dev, "debug_irq write error\n");
        ret = -EINVAL;
    }

    return ret;
}
```

```
/* Each client has this additional data */
struct mxt_data {
    /* use to track the IRQ status */
    atomic_t irq_processing;
}

static int mxt_acquire_irq(struct mxt_data *data)
{
    struct i2c_client *client = data->client;
    unsigned long irqflags = IRQF_TRIGGER_LOW;
    int error;

#if (LINUX_VERSION_CODE >= KERNEL_VERSION(4, 0, 0))
    irqflags = 0; // Use irqd_get_trigger_type() to acquire the DTS setting automatically,
otherwise you can ommit and hard coded it.
#endif

    dev_dbg(&data->client->dev, "enable irq(%d): irq_processing(%d) irqflags(0x%lx)\n",
        data->irq ? data->irq : client->irq, atomic_read(&data->irq_processing), irqflags);

    if (!data->irq) {
        atomic_set(&data->irq_processing, 1);
        error = devm_request_threaded_irq(&client->dev, client->irq,
                  NULL, mxt_interrupt, irqflags| IRQF_ONESHOT,
                  client->name, data);
        if (error) {
            atomic_set(&data->irq_processing, 0);
            dev_err(&client->dev, "Failed to register interrupt(%d) %d\n", client->irq, error);
            return error;
        }

        data->irq = client->irq;
    } else {
        atomic_inc(&data->irq_processing);
        enable_irq(data->irq);
    }

    return 0;
}
```

Code comparison

(Multi threaded processing)

```
mxt_update_seq_num_lock()
__mxt_read_reg_crc()
__mxt_write_reg_crc()
```

Lock the Seqnum update operation to avoid multi thread conflict.

```
static u8 mxt_update_seq_num_lock(struct mxt_data *data, bool reset_counter, u8 counter_value)
{
    mutex_lock(&data->i2c_lock);
    // ... Change Seqnum
    mutex_unlock(&data->i2c_lock);
}

static int __mxt_read_reg_crc(struct i2c_client *client,
                   u16 reg, u16 len, void *val, struct mxt_data *data, u8 flag)
{
    mutex_lock(&data->i2c_lock);

    // ... data transfer

    mutex_unlock(&data->i2c_lock);

}

static int __mxt_write_reg_crc(struct i2c_client *client, u16 reg, u16 length,
               const void *val, struct mxt_data *data)
{
    mutex_lock(&data->i2c_lock);

    // ... data transfer

    mutex_unlock(&data->i2c_lock);
}
```

Code comparison

(Flow chart/ Retrigen workaround)

Set `use_retrigen_workaround` to enable the retrigen bit and avoid last after reset.

Compatible the legacy kernel without irqflag query.

```
static int mxt_check_retrigen(struct mxt_data *data)
{

    struct i2c_client *client = data->client;
    int error;
    int val;
    int buff;
#if (LINUX_VERSION_CODE >= KERNEL_VERSION(4, 0, 0))
    unsigned long irqflags;
#endif

    data->use_retrigen_workaround = false;

    /*Iqnore when using level triggered mode */
#if (LINUX_VERSION_CODE >= KERNEL_VERSION(4, 0, 0))
    irqflags = irq_get_trigger_type(data->irq);
    if (irqflags & IRQF_TRIGGER_LOW) {
        dev_info(&client->dev, "Level triggered\n");
        return 0;
    } else {
        dev_info(&client->dev, "Get Irqflags 0x%lx, will check Retrigen mode\n", irqflags);
    }
#else
    //assume request_threaded_irq() default using IRQF_TRIGGER_LOW as trigger mode
    return 0;
#endif

    //  set Retrigen bit in T18

    dev_info(&client->dev, "RETRIGEN Enabled feature\n");
    data->use_retrigen_workaround = true;

    return 0;
}
```

```
static void mxt_proc_t6_messages(struct mxt_data *data, u8 *msg)
{
    /* Detect reset */
    if (status & MXT_T6_STATUS_RESET) {
        // recheck the retrign workaround
        if (data->use_retrigen_workaround) {
            mxt_check_retrigen(data);
        }

        complete(&data->reset_completion);
    }
}
```

Code comparison

(Flow chart/ Reset)

Align the mxt_reset() with Software mode and Hardware mode, but the parameter of `flag`. The reset command will alined with the Seqnum.

And for the `__hard_reset()`, need compatible gpio or gpiod operation.

```
// BIT(0) is resersed for the compatibility with maxtouch studio of /sys/
#define F_R_RSV BIT(0)
#define F_R_SOFT BIT(1)
#define F_R_HARD BIT(2)
#define F_R_WAIT BIT(3)
#define F_RST_SOFT  (F_R_SOFT | F_R_WAIT)
#define F_RST_HARD  (F_R_HARD | F_R_WAIT)
#define F_RST_ANY   (F_R_SOFT | F_R_HARD | F_R_WAIT)

static int mxt_reset(struct mxt_data *data, u8 flag)
{
    struct device *dev = &data->client->dev;
    int ret = 0;

    dev_info(dev, "Resetting device(%02X)\n", flag);

    mxt_disable_irq(data);

    INIT_COMPLETION(data->reset_completion);

    ret =__mxt_reset(data, flag);

    mxt_acquire_irq(data);

    if (!ret) {
        ret = mxt_wait_for_completion(data, &data->reset_completion,
                      MXT_RESET_TIMEOUT);
        if (ret) {
            dev_err(dev, "Wait for Resetting timeout(%d)\n", ret);
            return ret;
        }
    } else {
        dev_err(dev, "Resetting device failed(%d)\n", ret);
            return ret;
    }

    return 0;
}
```

```
static int __hard_reset(struct mxt_data *data, u8 flag)
{
    struct device *dev = &data->client->dev;

    dev_info(dev, "Resetting chip(H)\n");

    if (!data->reset_gpio) {
        return -EIO;
    }

    // Low level for asserting HW reset, this set whether active of `reset-gpios` setting in DTS
    // Note 1: if you using the non-standard kernel, there may be not be compatible.
    // So that, you could consider to switch the active level
    // Now we use directly gpio control(discard DTS setting): Low --- Reset active; High --- Chip working

    // Note 2: Please be wared of that, the maxtouch need special POR sequence that you must stretch Reset low before VDD raised to target voltage(~3.3v)
    // If you don't match this in Por, the chip have chance to halt. Assert the hardware reset only is not benifit for POR.

#if (LINUX_VERSION_CODE >= KERNEL_VERSION(4, 0, 0))
    gpiod_set_value(data->reset_gpio, true);    //Reset active
#else
    gpio_set_value(data->reset_gpio, 0);    //Reset active
#endif
    msleep(MXT_RESET_GPIO_TIME);

    /* After reset, need to update seq num to ZERO */
    mxt_update_seq_num_lock(data, true, 0x00);

#if (LINUX_VERSION_CODE >= KERNEL_VERSION(4, 0, 0))
    gpiod_set_value(data->reset_gpio, false);   //Reset active
#else
    gpio_set_value(data->reset_gpio, 1);    //Reset inactive
#endif

    // Wait for Reset completed by timeout
    if (flag & F_R_WAIT) {
        msleep(MXT_RESET_INVALID_CHG);
    }

    return 0;
}
```

Code comparison

(Flow chart/ Config update)

`mxt_update_crc()` with wait whatever the crc or non crc accessing.

`mxt_update_cfg():`

- Remove the redundant code mxt_init_t7_power_cfg() / mxt_check_retrigen()

- Return value meaningful:

Return 0: skip update
Return 1: updated
Other value: update failed

- Support config file name assigned in `update_cfg` interface

`mxt_clear_cfg()` Never do backup after clear, it's dangerous to frozen the touch panel

```c
static void mxt_update_crc(struct mxt_data *data, u8 cmd, u8 value)
{
	/*
	 * On failure, CRC is set to 0 and config will always be
	 * downloaded.
	 */

	INIT_COMPLETION(data->crc_completion);

	mxt_t6_command(data, cmd, value, true);

	/*
	 * Wait for crc message. On failure, CRC is set to 0 and config will
	 * always be downloaded.
	 */
	mxt_wait_for_completion(data, &data->crc_completion, MXT_CRC_TIMEOUT);
}
```

Code comparison

(Flow chart/ FW update)

`mxt_load_fw()` set Seqnum Zero when enter bootloader mode

Unmark `in_bootloader` on when irq disabled that avoid interrupt handler in processing after firmware updated.

`mxt_update_fw_store()`

- support firmware name assignment
- Recovery if firmware update failed or skipped
- Remove redundant code of crc_enable mark and retrigen bit check

```
static int mxt_load_fw(struct device *dev, const char *fn)
{
    mxt_update_seq_num_lock(data, true, 0x00);

disable_irq:
    mxt_disable_irq(data);
release_firmware:
    release_firmware(fw);

    if (!ret) {
        // We move here that after firmware finished, the interrupt may be working to processing
message,
        // So we should keep irq disable before assert the exit of bootloader
        data->in_bootloader = false;
    }


}


static ssize_t mxt_update_fw_store(struct device *dev,
                    struct device_attribute *attr,
                    const char *buf, size_t count)
{
    error = mxt_update_file_name(dev, &file_name, buf, count);
    if (error) {
        dev_err(dev, "Failed get file name: %s\n", buf);
        return error;
    }

    error = mxt_load_fw(dev, file_name);
    if (error) {
        dev_err(dev, "The firmware update failed(%d)\n. IRQ disabled.", error);
        mxt_disable_irq(data);

        dev_err(dev, "Executing hardware reset");
        // Not the `irq` should be disabled to call __mxt_reset() for the Seq num synchronization
        count = error;
    } else {
        dev_info(dev, "The firmware update succeeded, Reset\n");
        msleep(MXT_FW_FLASH_TIME);
    }

    kfree(file_name);

    __mxt_reset(data, F_RST_ANY);   // Any reset

}
```

Code comparison

(T7 check)

`mxt_init_t7_power_cfg()`

set T7 power only if the latest T7 config is invalid. We will make a copy if T7 config is verified.

```
static int mxt_init_t7_power_cfg(struct mxt_data *data)
{
    struct device *dev = &data->client->dev;
    struct t7_config t7_cfg;
    int error;
    bool retry = false;

recheck:
    error = mxt_read_reg_auto(data->client, data->T7_address,
            sizeof(t7_cfg), &t7_cfg, data);

    if (error)
        return error;

    if (t7_cfg.active == 0 || t7_cfg.idle == 0) {
        if (!retry) {
            dev_info(dev, "T7 cfg zero, resetting\n");
            retry = true;
            goto recheck;
        } else {
            dev_info(dev, "T7 cfg zero after reset, overriding\n");
            if (data->t7_cfg.active == 0 || data->t7_cfg.idle == 0) {   // Try lastest t7_cfg
first
                data->t7_cfg.active = 20;
                data->t7_cfg.idle = 100;
            }
            return mxt_set_t7_power_cfg(data, MXT_POWER_CFG_RUN);
        }
    } else {
        memcpy(&data->t7_cfg, &t7_cfg, sizeof(t7_cfg));
    }

    dev_info(dev, "Initialized power cfg: ACTV %d, IDLE %d\n",
        data->t7_cfg.active, data->t7_cfg.idle);
    return 0;
}
```

Code comparison

(T15 button processing)

Experimental to support 2 instance of T15 object.

`mxt_proc_t15_messages`() processing T15 event as each instance
`mxt_read_t15_num_keys_inst`() readout the key count of T15 instance

```
static void mxt_proc_t15_messages(struct mxt_data *data, u8 *msg)
{
    struct input_dev *input_dev = data->input_dev;
    struct device *dev = &data->client->dev;
    int key;
    bool curr_state, new_state;
    bool sync = false;
    unsigned long keystates = (msg[2] | (u16)msg[3] << 8);
    unsigned int t15_num_keys;
    int id;
    u8 offset;

    id = msg[0] - data->T15_reportid_min;
    if (id == 0) {
        // Primary instance - using keystates low bits
        offset = 0;
        t15_num_keys = min(data->t15_num_keys_inst0, data->t15_num_keys);
    } else if (id == 1) {
        // Second instance - using keystates high bits
        offset = data->t15_num_keys_inst0;
        keystates <<= offset;
        t15_num_keys = data->t15_num_keys;
    } else {
        dev_err(dev, "Unknown T15 %d\n", id);
        return;
    }

    dev_dbg(dev, "T15 [%d] status (0x%lx - 0x%lx)\n", id, data->t15_keystatus, keystates);

    for (key = offset; key < t15_num_keys; key++) {
           // … processing key event
    }

    if (sync) {
        input_sync(input_dev);
    }
}

static int mxt_read_t15_num_keys_inst(struct mxt_data *data, u8 instance, unsigned int* nks)
{
   // read out the key num of T15 instance
}
```

Code comparison

(Flow chart/ Resync)

Re-achieve the mxt_resync_comm() function.

Call Resync only when:

mxt_read_and_process_messages()
mxt_process_messages_t44_t144()

Removed its callee in other place.

```
static void mxt_proc_t6_messages(struct mxt_data *data, u8 *msg)
{
    /* Detect reset */
    if (status & MXT_T6_STATUS_RESET) {
        // recheck the retrign workaround
        if (data->use_retrigen_workaround) {
            mxt_check_retrigen(data);
        }

        complete(&data->reset_completion);
    }
}
```

Code comparison

(Flow chart/ Message processing)

1. The old processing had 1 byte memory leakage and fixed.
2. Simplify the message processing:
- Read T44/144 to get message count
- Call `mxt_read_and_process_messages()` to read and process each message in T5

```
static irqreturn_t mxt_process_messages_t44_t144(struct mxt_data *data)
{
    struct device *dev = &data->client->dev;
    int ret;
    u16 address;
    u8 count, num_left;

    /* Read T44 and T5 together for legacy devices */
    /* For new HA parts, read only T144 count */
    if (data->T144_address) {
        address = data->T144_address;
    } else {
        address = data->T44_address;
    }

    ret = mxt_read_reg_auto(data->client, address,
            data->msg_count_size, data->msg_buf, data);

    num_left = count;

    /* Process remaining messages if necessary */
    if (num_left) {
        ret = mxt_read_and_process_messages(data, num_left);
    }
}
```

Code comparison

(Input device registration)

Remove the mxt_init_secondary_input() and combine the code mxt_initialize_input_device().
The 2nd parameter will decide which input device.

Remove the BTN_TOUCH, ABS_X, ABS_Y for compatible issue.
Set EV_ABS flag for compatible issue

Device open and close only useful for CONFIG_ACPI marked

Add `t15_num_keys_inst0` to store instance 0 key num

```
static struct input_dev * mxt_initialize_input_device(struct mxt_data *data, bool primary)
{
#ifdef CONFIG_ACPI
    input_dev->open = mxt_input_open;
    input_dev->close = mxt_input_close;
#endif
#if (LINUX_VERSION_CODE < KERNEL_VERSION(4, 0, 0))
    set_bit(EV_ABS, input_dev->evbit);
#endif

#ifdef CONFIG_INPUT_DEVICE2_SINGLE_TOUCH
    // For single touch //
    input_set_capability(input_dev, EV_KEY, BTN_TOUCH);
    input_set_abs_params(input_dev, ABS_X, 0, data->max_x, 0, 0);
    input_set_abs_params(input_dev, ABS_Y, 0, data->max_y, 0, 0);
#endif

    /* If device has buttons we assume it is a touchpad */
    if (primary && data->t19_num_keys) {
#if (LINUX_VERSION_CODE >= KERNEL_VERSION(4, 0, 0))
        mt_flags = INPUT_MT_POINTER;
#endif
    } else {
#if (LINUX_VERSION_CODE >= KERNEL_VERSION(4, 0, 0))
        mt_flags = INPUT_MT_DIRECT;
#endif
    }

    /* For multi touch */
#if (LINUX_VERSION_CODE >= KERNEL_VERSION(4, 0, 0))
    error = input_mt_init_slots(input_dev, num_mt_slots, mt_flags);
#else
    __set_bit(INPUT_PROP_DIRECT, input_dev->propbit);
    error = input_mt_init_slots(input_dev, num_mt_slots);
#endif

    /* For T15 Key Array */
    if (data->T15_reportid_min) {
        data->t15_keystatus = 0;
        if (data->t15_num_keys) {
            error = mxt_read_t15_num_keys_inst(data, 0, &data->t15_num_keys_inst0);
            if (error) {
                // Set key to num DTS defined keys.
                data->t15_num_keys_inst0 = data->t15_num_keys;
            }

            for (i = 0; i < data->t15_num_keys; i++) {
                input_set_capability(input_dev, EV_KEY,
                        data->t15_keymap[i]);
            }
        }
    }
}
```

Code comparison

(Readable)

Put the HA ID check in function.
mxt_lookup_ha_chips()

__mxt_read_reg_crc()
Use flag to control the Seqnum and CRC check instead of the breakthrough accessing of T144/T5

mxt_write_reg_auto()
Using the function wrapper to decide whether it's CRC operation

```
static bool mxt_lookup_ha_chips(const struct mxt_info *info)
{
    u8 family_id;
    u8 variant_id;
    bool is_ha = false;

    if (!info) {
        return false;
    }

    family_id = info->family_id;
    variant_id = info->variant_id;

    switch (family_id) {
        case 0xA6:
            if (variant_id == 0x14) {
                // "336UD-HA"
                is_ha = true;
            }
        break;
        default:
            ;
    }
    return is_ha;
}

#define F_R_SEQ BIT(0)
#define F_R_CRC BIT(1)
static int __mxt_read_reg_crc(struct i2c_client *client,
                   u16 reg, u16 len, void *val, struct mxt_data *data, u8 flag);


static int mxt_write_reg_auto(struct i2c_client *client, u16 reg, u16 length,
               const void *val, struct mxt_data *data)
{
    if (data->crc_enabled) {
        return __mxt_write_reg_crc(client, reg, length, val, data);
    } else {
        return __mxt_write_reg(client, reg, length, val);
    }
}
```

Code comparison

(Readable)

Use `mxt_config_cb()` callback to initialization device instead of redundant code

```
static int mxt_initialize(struct mxt_data *data)
{
    while (1) {
        error = mxt_read_info_block(data);
    }

    error = mxt_acquire_irq(data);
    if (error) {
        dev_err(&client->dev, "Acquire irq %d failed(%d)\n", data->irq, error);
        return error;
    }

    if (true){
        /* As built-in driver, root filesystem may not be available yet */
        error = request_firmware_nowait(THIS_MODULE, true, MXT_CFG_NAME,
                                &client->dev, GFP_KERNEL, data,
                                mxt_config_cb);
        if (error) {
            dev_warn(&client->dev, "Failed to invoke firmware loader: %d\n",
                error);
        }
    } else {
        mxt_configure_objects(data, NULL);
    }

    return 0;
}
```

Code comparison

(Selftest)

Added T10/T25 message processing
Added selftest interface in Sys

```
static void mxt_proc_t10_messages(struct mxt_data *data, u8 *msg)
{
    struct device *dev = &data->client->dev;
    u8 status = msg[1];
    u8 cmd = msg[2];

    /* Output debug if status has changed */
    dev_info(dev, "T10 Status 0x%2x CMD %d Info: %02x %02x %02x\n",
        status,
        cmd,
        msg[3],
        msg[4],
        msg[5]);

    /* Save current status */
    memcpy(data->selftest_msg, msg, sizeof(data->selftest_msg));
}

static void mxt_proc_t25_messages(struct mxt_data *data, u8 *msg)
{
    struct device *dev = &data->client->dev;
    u8 status = msg[1];
    /* Output debug if status has changed */
    dev_info(dev, "T25 Status 0x%2x Info: %02x %02x %02x %02x %02x\n",
        status,
        msg[2],
        msg[3],
        msg[4],
        msg[5],
        msg[6]);

    /* Save current status */
    memcpy(data->selftest_msg, msg, sizeof(data->selftest_msg));
}

static int mxt_set_selftest(struct mxt_data *data, u8 cmd, bool wait)
{
   …
}

static ssize_t mxt_selftest_show(struct device *dev,
                    struct device_attribute *attr, char *buf)
{
    …
}

static ssize_t mxt_selftest_store(struct device *dev,
    struct device_attribute *attr, const char *buf, size_t count)
{
    …
}
```

Code comparison

(Resource recycling)

`mxt_free_second_input_device()` Free the second input device when need

msg_buf: malloc enough memory avoid memory overflow

dev_id_buf: free memory after resync

`mxt_remove()` release the resource if applied

```
static void mxt_free_second_input_device(struct mxt_data *data)
{
    if (data->input_dev_sec) {
        input_unregister_device(data->input_dev_sec);
        data->input_dev_sec = NULL;
    }
}


static int mxt_parse_object_table(struct mxt_data *data,
                  struct mxt_object *object_table)
{

    data->msg_buf = kcalloc(data->max_reportid,
                data->T5_msg_size + data->msg_count_size, GFP_KERNEL);
}


static int mxt_resync_comm(struct mxt_data *data)
{

   // ….Resync process
err_free_mem:
    if (dev_id_buf) {
        kfree(dev_id_buf);
    }

    if (info) {
        if (info != data->info) {
            kfree(info);
        }
    }
}
```

```
static int mxt_remove(struct i2c_client *client)
{
    struct mxt_data *data = i2c_get_clientdata(client);

    dev_info(&client->dev, "ATMEL MaXTouch Driver Removed\n");

    if (data) {
        mxt_debug_msg_remove(data);
        mxt_sysfs_remove(data);

        mxt_free_irq(data);
        atmel_mxt_ts_teardown_debugfs(data);

        mxt_free_input_device(data);
        mxt_free_second_input_device(data);
        mxt_free_object_table(data);
        mxt_free_device_properties(data);

        devm_kfree(&client->dev, data);
        i2c_set_clientdata(client, NULL);
    }

    return 0;
}
```

The End